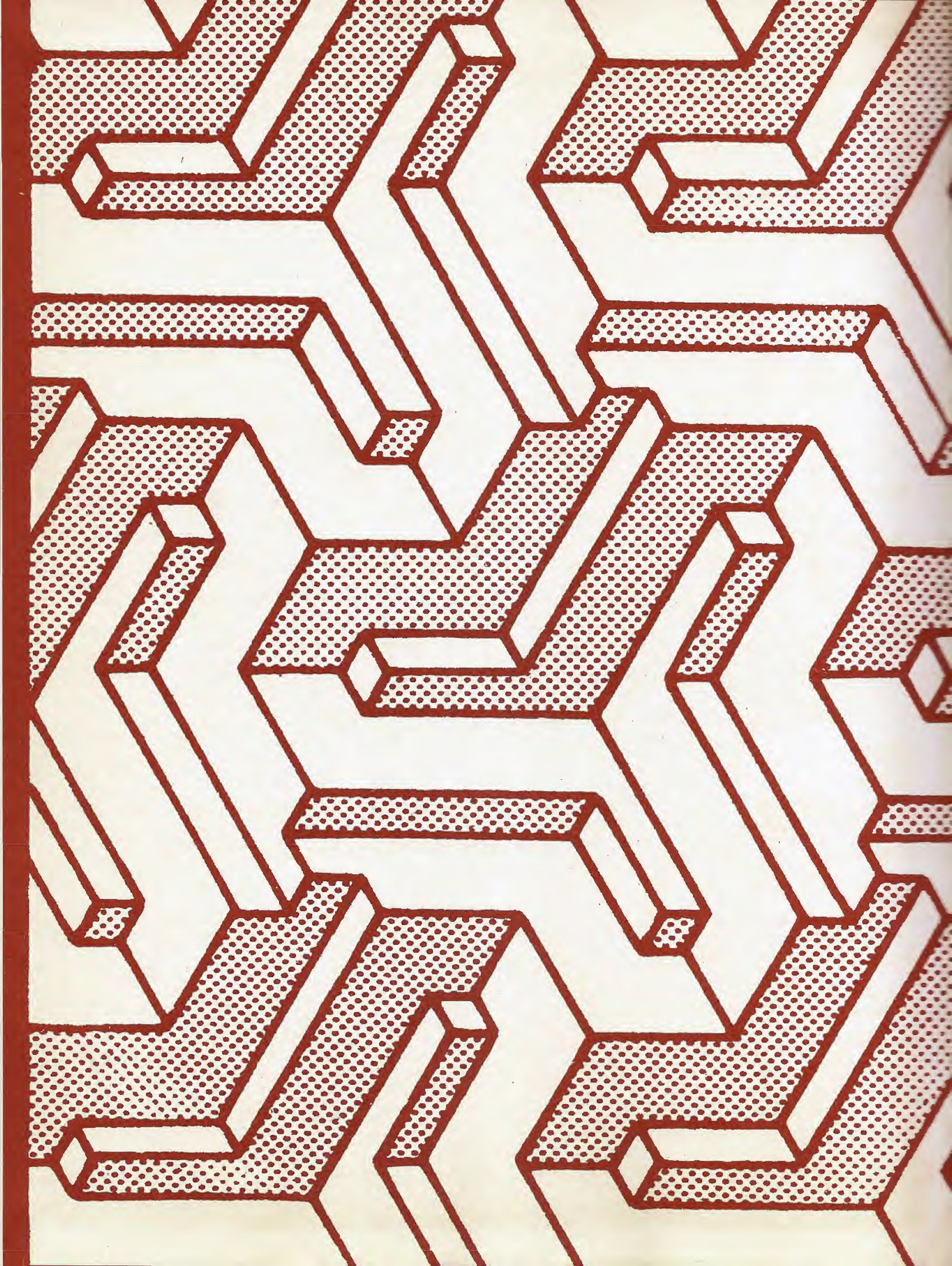
GRAN ENCICLOPEDIA INFORMATICA

CURSO DE BASIC/ 1

EDICIONES NUEVA LENTE



GRAN ENCICLOPEDIA INFORMATICA

EDICIONES NUEVA LENTE


break

SUMARIO

Una publicación:

Ediciones Nueva Lente, S. A.

Director editor: MIGUEL J. GOÑI

Director de producción: SANTOS ROBLES.

Director de la obra: FRANCISCO LARA.

Colaboradores: PL/3 - MANUEL MUÑOZ - ANGEL MARTINEZ - MIGUEL DE ROSENDO - DAVID SANTAOLALLA - SANTIAGO RUIZ - LUIS COCA - MIGUEL ANGEL VILA - MIGUEL ANGEL SANCHEZ VICENTE ROBLES.

Diseño: BRAVO/LOFISH.

Maquetación: JUAN JOSE DIAZ SANCHEZ.

Ilustración: JOSE OCHOA.

Fotografía: (Equipo Gálata) ALBINO LOPEZ y EDUARDO AGUDELO.

Ediciones Nueva Lente, S. A.:
Dirección y Administración:
Benito Castro, 12. 28028 Madrid. Tel.: 245 45 98.

Números atrasados y suscripciones:
Ediciones Ingelek, S. A.
Plaza de la Rep. Ecuador, 2 - 1.°. 28016 Madrid.
Tel.: 250 58 20.

Plan general de la obra:
18 tomos monográficos de aparición quincenal.

Distribución en España:
COEDIS, S. A. Valencia, 245. Tel.: 215 70 97.
08007 Barcelona.

Delegación en Madrid:
Serrano, 165. Tel.: 411 11 48.

Distribución en Argentina:
Capital: AYERBE
Interior: DGP

Distribución en Chile: Alfa Ltda.

Distribución en México:
INTERMEX, S. A.
Lucio Blanco, 435
México D.F.

Distribución en Uruguay:
Ledian, S. A.

Edita para Chile:
PYESA
Doctor Barros Borgoño, 123
Santiago de Chile

Importador exclusivo Cono Sur:
CADE, SRL. Pasaje Sud América, 1532.
Tel.: 21 24 64. Buenos Aires - 1.290. Argentina.

Fotomecánica: Ochoa, S. A.
Miguel Yuste, 32. 28037 Madrid.

Impresión: Gráficas Reunidas, S. A.
Avda. de Aragón, 56. 28027 Madrid.

ISBN de la obra: 84-7534-184-5.

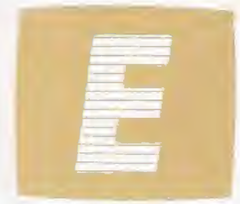
ISBN del tomo 8: 84-7534-205-1

Printed in Spain
Depósito legal: M. 27.605-1986

Precio de venta al público en Canarias, Ceuta y Melilla: 940 ptas.
Enero 1987

Introducción al BASIC

Una visión general de este popular lenguaje



l lenguaje de programación más común en el campo de los ordenadores personales es el BASIC. Su nombre está construido a partir de las iniciales de *Beginner's All-Purpose Symbolic Instruction Code:* Código de instrucciones simbólicas de uso general para principiantes. Toda una definición del objetivo que motivó la creación de este popular lenguaje.

A la hora de desarrollar un lenguaje de programación polivalente y de fácil uso por los programadores noveles, los creadores del BASIC se inspiraron en dos lenguajes de alto nivel muy popularizados: el FORTRAN y el ALGOL.

Como tal lenguaje, el BASIC tiene su propio vocabulario y sus reglas sintácticas; y por supuesto, las particularidades inherentes a los lenguajes destinados al diálogo con los ordenadores.

Está concebido para ser una herramienta a la hora de utilizar el ordenador; en su mayor parte consta de órdenes que la máquina ha de ejecutar.



¿Cómo hay que hablar con un ordenador? ¿Es posible establecer una comunicación sin recurrir al código máquina? El BASIC constituye un claro ejemplo de lenguaje de alto nivel capaz de diluir la distancia entre el usuario y la máquina.

El arte de dialogar en BASIC

Si el ordenador sólo es capaz de entender los arcanos del cero y el uno, ¿cómo es posible que los ordenadores entiendan el repertorio de frases, muy parecidas al lenguaje ordinario, que pueden construirse con el vocabulario de un lenguaje evolucionado como el BASIC?

Nada más sencillo... ¿Para qué están los traductores? Dada la enorme capacidad de trabajo del ordenador, también podrá encargarse de convertir nuestras frases en BASIC a las secuencias de ceros y unos de su propio lenguaje interno.

La máquina aguarda a que se la instruya, y no hay óbice que impida que el primer programa que reciba sea, precisamente, un traductor. Como ya sabemos, los traductores de lenguajes pueden ser de dos tipos: *intérpretes* o *compiladores.*

Cabe recordar que la actuación del programa traductor tendrá un total paralelismo con la realidad cotidiana. Por ejemplo, si un científico extranjero quiere dar una conferencia y ni él sabe castellano, ni el público conoce su idioma, caben dos soluciones: que un intérprete vaya traduciendo su disertación, paso a paso, o que se reparta una hoja con el contenido ya traducido.

En efecto, el ordenador no entiende el lenguaje BASIC directamente, sino que antes de asimilar su contenido, debe traducirlo a su propio código (el lenguaje máquina). Tal como ya hemos señalado, las comunicaciones con la máquina en BASIC son, con frecuencia, interactivas; de ahí que el traductor de BASIC a lenguaje máquina sea, habitualmente, un intérprete de BASIC. No obstante, cuando el BASIC se utilice para programar tareas complejas, cuya posterior ejecución debe ser rápida, es obvio que resultará más adecuado sustituir el intérprete por un compilador BASIC.

La ejecución de un programa compilado (traducido en bloque) es bastante más rápida que la ejecución de un programa canalizado a través de un intérprete. Volviendo al ejemplo anterior, la conferencia se prolongará durante más tiempo si un intérprete debe traducir frase a frase la disertación del conferenciante; por el contrario, si el texto de la conferencia se traduce de una sola vez (tarea del compilador) y se entrega traducido a los presentes, la duración de la conferencia será más breve.

A la hora de establecer un primer contacto con el mundo de la programación en BASIC, o incluso acometer la confección de programas, el intérprete BASIC se revela como un eficaz auxiliar. Entre sus funciones están las de detectar errores y comunicarlos al usuario, además de facilitar las tareas de corrección y depuración de los programas.

Instrucciones y programas

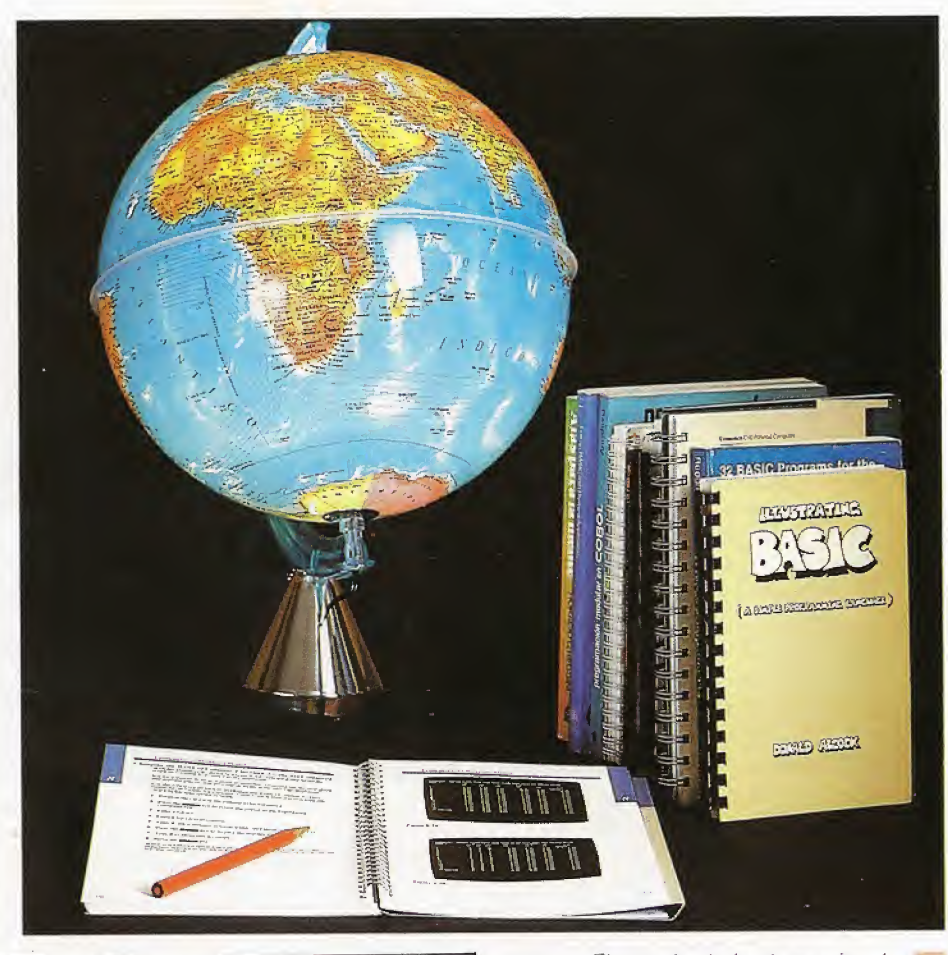
Al igual que cualquier lenguaje humano, el BASIC se rige por un conjunto de normas semánticas y reglas sintácti-

cas cuya puesta en práctica permite construir mensajes organizados. La base del lenguaje se encuentra en su vocabulario o repertorio de palabras, a través de las cuales es posible expresar cualquier acción programable. El vocabulario del BASIC está constituido por palabras del idioma inglés. Un vocabulario bastante más simple y limitado que el de cualquier lenguaje humano, pero suficiente para construir las órdenes destinadas a la máquina. Tal como ocurre en los diálogos humanos, la unidad de comunicación se encuentra en la frase, o conjunto de palabras que expresan una determinada acción. En el caso del BASIC, la unidad de comunicación se denomina *instrucción*.

El objeto de una instrucción es «educar» al ordenador para que éste realice una tarea específica.

Agrupando un cierto número de frases, es posible expresar una actividad completa con todos sus matices. Lo mismo ocurre en el ámbito de los lenguajes de programación al asociar un determinado número de instrucciones destinadas a la máquina. El conjunto organizado de instrucciones que definen una tarea completa recibe el nombre de *programa*.

La estructura de las frases del len-



El mundo de los lenguajes de programación guarda un gran paralelismo con el de los lenguajes humanos. Existen idiomas de mayor relevancia, como el inglés, que constituyen verdaderos patrones de entendimiento universal. Esta característica se hace extensiva al ámbito de las máquinas, donde el BASIC se revela como el lenguaje más generalizado y universal.

Una vez conectado el ordenador, intentamos el primer diálogo. Con sorpresa observamos que no es capaz de entender nuestro idioma. En efecto, las máquinas programables tienen su propio lenguaje —el BASIC es el primordial—, que el usuario debe aprender y utilizar para hacer posible la comunicación.

guaje común se mantiene, con ligeras salvedades, en las frases o instrucciones que confirman un programa en lenguaje BASIC. Un verbo es quien define la acción a realizar por el sujeto; análogamente, un *comando* es el que indica la acción que debe realizar el ordenador con los datos.

En una instrucción BASIC intervienen dos zonas:

- Comando
- Argumento

El comando (semejante al verbo de una frase convencional) expresa la acción. Este puede coincidir con una simple orden, desprovista de argumento, dirigida a la máquina. Por ejemplo:

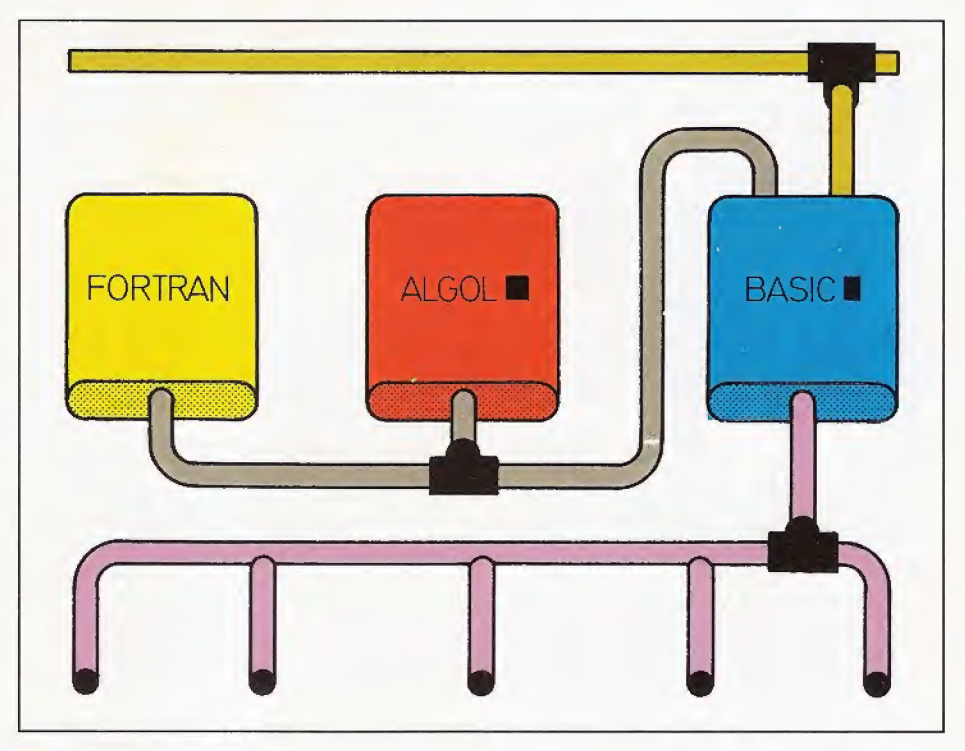
STOP: Detener la secuencia de ejecución.
NEW: Borrar por completo el programa en curso.
RUN: Ejecutar el programa.

Cuando la acción afecta a un dato, a un grupo de datos, o a un dispositivo asociado al ordenador, el comando se acompaña de una segunda zona denominada *argumento*. Si la instrucción incluye zonas, el comando suele recibir el nombre de *sentencia*.

```
LET A=20
PRINT 430
GOTO 25
```

Todas estas instrucciones incluyen una sentencia (comando) seguida por un argumento que aporta el dato o datos implicados en la ejecución de la orden. Por ejemplo, la primera instrucción apuntada (LET A=20) insta a la máquina para que asigne el valor 20 a la variable A. La segunda (PRINT 430) comunica al ordenador que lleve a la pantalla el dato 430; mientras que GOTO 25 ordena un salto a la instrucción número 25.

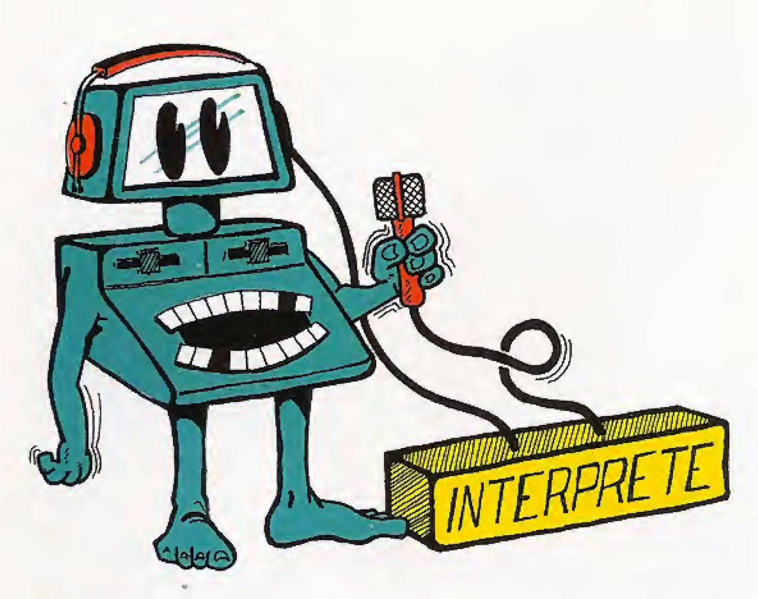
Dentro de la zona de argumento caben muy diversos tipos de datos y con distintas expresiones: valores numéricos, palabras o grupos de caracteres alfanuméricos, variables, referencias a dispositivos periféricos... Habitualmente, el argumento contiene varios datos relacionados entre sí por *operadores* (suma, resta, multiplicación, igual-



El BASIC es el lenguaje de programación más popular de nuestros días. Sus raíces parten de otros dos lenguajes informáticos: el FORTRAN y el ALGOL.



Son muy diversos los dialectos del BASIC que coexisten en la actualidad. La mayor parte de los fabricantes incluye en sus equipos un intérprete BASIC con ciertas peculiaridades en su vocabulario y sintaxis.



El BASIC es un lenguaje que suele utilizarse en un ámbito de comunicación interactiva. De ahí que, normalmente, la traducción del lenguaje BASIC se encomiende a un programa «intérprete».

dad...), o afectados por *funciones,* ya sean matemáticas o de cualquier otro tipo autorizado por el lenguaje BASIC.

Instrucciones directas e indirectas

En un diálogo caben tanto frases sueltas, que dan pie a una respuesta inmediata, como mensajes que agrupan a un determinado número de frases. Esta es una realidad aplicable también al lenguaje BASIC. El usuario puede dirigirse al ordenador con una simple instrucción y aguardar su respuesta. O bien puede comunicarse recurriendo a una secuencia de instrucciones o programa.

En ambos casos las instrucciones son las mismas, aunque se emplean de distintas forma. Cuando la instrucción se utiliza de forma independiente, para que la máquina la ejecute y curse una respuesta inmediata, recibe el nombre de *instrucción directa.* Por el contrario, si ésta forma parte de un programa y se encuentra precedida por un número de orden que denota su situación dentro del mismo, se estará formulando como *instrucción indirecta.*

Al accionar la tecla RETURN tras una instrucción en modo directo, el ordenador la asimilará y ejecutará sin mayor dilación. Por ejemplo, si se introduce la instrucción LET A=20, al pulsar la tecla RETURN el ordenador asignará de inmediato el valor 20 a la variable A.

De forma análoga, tras dar la orden RETURN, después de introducir PRINT 35 aparecerá en la pantalla el dato indicado (35).

Ambas instrucciones pueden agruparse dentro de un programa, lo que equivale a utilizarlas de modo indirecto. En este caso, cada instrucción debe estar precedida por un número (número de línea) que servirá al ordenador para discernir en qué orden debe ejecutarlas. Por ejemplo:

```
10 LET A=250
20 PRINT A
■
```

Las instrucciones utilizadas en modo indirecto, ya no son ejecutadas por el ordenador al accionar la tecla RETURN que pone fin a cada línea. Estas pasan a constituir un programa que será ejecutado en bloque al introducir la orden al efecto RUN.

```
10 LET A=250
20 PRINT A
RUN

250
■
```

En el ejemplo, al recibir la orden RUN, el ordenador ha ejecutado ambas instrucciones en el orden indicado por el número de línea que las acompaña. Al ejecutar la instrucción 10 (LET A=250) se asigna a la variable A el valor 250. Acto seguido, se ejecuta la instrucción 20 que ordena imprimir el valor de la variable A.

PRINT

Imprime en la pantalla los mensajes o el valor de las expresiones que aparecen en la zona de argumento.

Formato: (Número de línea) PRINT<*expresión 1*>{[,][;]}<*expresión 2*>...

Ejemplos: PRINT 25
10 PRINT "GEI"
35 PRINT A,C$,"CURSO DE BASIC"

Notación utilizada en el formato:

- <> : Los textos y expresiones encerradas por los símbolos «menor» o «mayor» son aportación del usuario.
- [] : Los elementos encerrados entre corchetes son opcionales.
- { } : Las llaves delimitan a los elementos alternativos.

Aspecto de un programa BASIC

Como ya hemos visto, un programa es una secuencia ordenada de instrucciones que definen una tarea completa. Las instrucciones se distribuyen en líneas sucesivas, cada una de ellas precedida por un número entero que señala el orden en el que será ejecutada por el ordenador.

Los números de línea deben seguir un orden creciente. De esta forma, la ejecución de la secuencia de instrucciones hará que el ordenador realice las sucesivas operaciones que conducirán al resultado final.

Un programa redactado en lenguaje BASIC presenta un estructura semejante a la que sigue:

```
10 INSTRUCCION 1
20 INSTRUCCION 2
30 INSTRUCCION 3
40 INSTRUCCION 4: INSTRUCCION 5
50 INSTRUCCION 6
55 END
█
```

```
10 REM ***************************************************************************************
20 REM **                    POSICIONAMIENTO DEL CURSOR                     **
30 REM **                          EN PANTALLA                              **
40 REM ***************************************************************************************
50 CLS
60 LET X=10: LET Y=15
70 PRINT AT (X,Y); "*"
80 PRINT AT (X,Y-1); " "
90 PRINT AT (X,Y+1)); " "
100 LET B$=INKEY$: IF B$="" THEN GOTO 100
110 IF B$="W" THEN LET Y=Y-1
120 IF B$="Z" THEN LET Y=Y+1
130 IF B$="A" THEN LET X=X-1
140 IF B$="S" THEN LET X=X+1
150 IF X>30 THEN LET X=30
160 IF X<0 THEN LET X=0
170 IF Y<1 THEN LET Y=1
180 IF Y>20 THEN LET Y=20
190 GOTO 70
```

Aspecto de un programa en lenguaje BASIC.

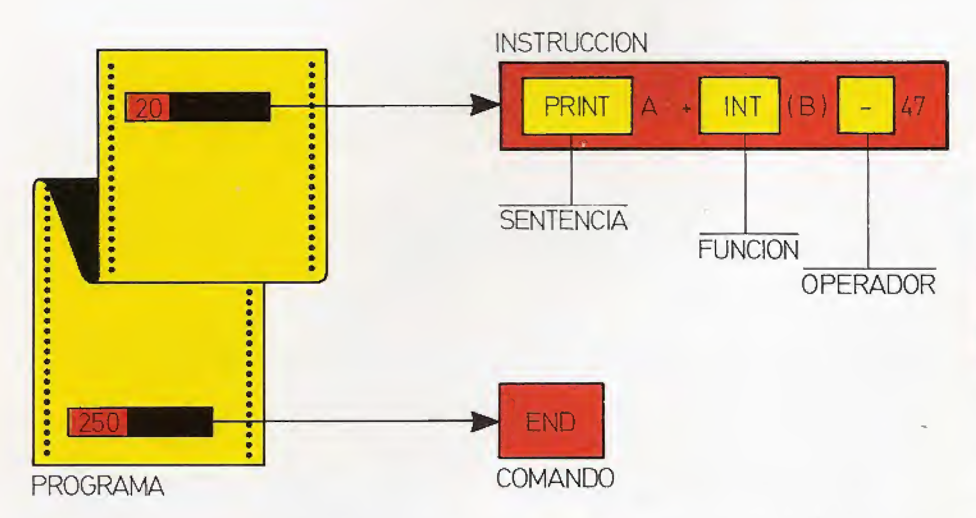
El número de línea instruye al ordenador sobre el orden en el que debe ejecutar el programa. Cabe observar que, excepto en la última línea de instrucción, se han numerado líneas a intervalos de 10. ¿Por qué...? En la práctica, el primer intento de escritura de un programa muy raramente se ve coronado por el éxito; lo habitual es que queden en el olvido algunas instrucciones. Numerando las líneas con un cierto intervalo, queda abierta la posibilidad de añadir nuevas instrucciones en el lugar oportuno, utilizando números de líneas intermedios. Recordemos que el ordenador ejecutará el programa atendiendo estrictamente a un orden numérico creciente; así pues, si hemos olvidado una instrucción intermedia entre las instrucciones 2 y 3 no tenemos por qué reescribir el programa completo, sino que bastará simplemente con escribir al final del programa la citada instrucción con el número de línea «a propósito», por ejemplo: 24. El intérprete BASIC se ocupará de que las instrucciones añadidas posteriormente se sitúen en el lugar que les corresponda dentro del programa, atendiendo a su número de línea.

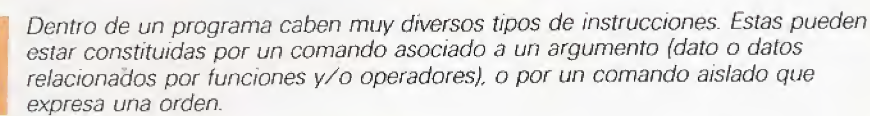
A la hora de escribir un programa, la orden RETURN tiene la misma importancia que «el retroceso de carro» en el caso de una máquina de escribir. Al dar por concluida la escritura de una línea, resulta imperativo dar la orden RETURN. El intérprete BASIC responderá de inmediato, mostrando en la pantalla el cursor o «indicador de presencia»; éste señalará la posición a partir de la que se escribirá la próxima instrucción.

Algunos intérpretes BASIC permiten la escritura de dos o más instrucciones en una misma línea de programa. Las diversas instrucciones presentes en la misma línea se separan por medio de un carácter denominado «separador de instrucción». En el ejemplo propuesta (línea 40) las instrucciones 4 y 5 se separan por medio del signo ":".

La necesidad de instruir al ordenador con toda suerte de precisiones, llega hasta el punto que es preciso señalar el final del programa. De ahí que la última línea del programa esté ocupada por una instrucción BASIC especializada en tal menester: la orden END.

Las instrucciones BASIC recuerdan a las fórmulas matemáticas. Una semejanza que apunta la utilidad del BASIC para resolver problemas científicos. A pesar de ello, el campo de acción de este lenguaje de alto nivel cubre todo tipo de disciplinas, desde los juegos y el aprendizaje hasta las tareas de gestión. Su naturaleza de lenguaje interactivo —el usuario recibe una respuesta instantánea a sus comunicaciones—, ha





Dentro de un programa caben muy diversos tipos de instrucciones. Estas pueden estar constituidas por un comando asociado a un argumento (dato o datos relacionados por funciones y/o operadores), o por un comando aislado que expresa una orden.

Una breve historia del BASIC

El lenguaje de programación BASIC (Beginner's All-Purpose Symbolic Instruction Code) nació en 1964 en el Dartmouth College, de la mano de John G. Kemeny y Thomas Kurtz, y fue concebido como un lenguaje interactivo, polivalente y de fácil aprendizaje y empleo.

En un principio fue normalizado por el organismo ANSI (American National Standards Institute) y de esta normalización parten las líneas originales del BASIC. Más tarde, surgió toda una gran familia de dialectos que cada vez se fueron desviando más y más del lenguaje original.

En 1977, la empresa americana Microsoft desarrolló un dialecto que pretendía unificar criterios. Rápidamente fue aceptado por varios fabricantes de ordenadores como Tandy, Apple, Commodore...

El gran boom del BASIC ha llegado con la irrupción de los microordenadores, con la gran ventaja de su precio, que los ha hecho asequibles a cualquier bolsillo. Pero hay que señalar que en un principio el BASIC fue adoptado por los sistemas comerciales de tiempo compartido. De éstos es de donde viene la popularidad del BASIC.

En la década de los ochenta, el BASIC se ha constituido en el lenguaje de programación más utilizado. Aunque se habla de varios lenguajes como futuros sustitutos del BASIC, lo cierto es que ninguno amenaza seriamente la posición privilegiada que éste mantiene en el campo del ordenador personal.

contribuido a que el BASIC sea el lenguaje más utilizado y popular.

El comando PRINT

La primera impresión de que el ordenador es una máquina con capacidad práctica, la obtenemos al observar la pantalla repleta de mensajes y dibujos. De ahí que uno de los primeros objetivos de todo aquel que empieza a programar sea, precisamente, escribir algo en la pantalla.

El lenguaje BASIC dispone de varios comandos especializados en esta función; entre ellos, el más importante es PRINT. Utilizándolo, en modo directo o indirecto, pueden llevarse a la pantalla los números, letras o palabras que desee el usuario.

El formato más simple de una instrucción PRINT es el que incluye el mencionado comando, seguido por un texto encerrado entre comillas. Por ejemplo::

```
PRINT "BASIC"
PRINT "M-6727"
PRINT "LENGUAJE DE PROGRAMACION"
```

Al utilizar la instrucción PRINT en modo directo (sin número de línea), el ordenador llevará a la pantalla el texto encerrado entre comillas, en el instante en el que se dé por concluida la instrucción con la orden RETURN.

```
PRINT "BASIC" (RT)
BASIC
■
```

(RT): Acción sobre la tecla RETURN.

Desde luego, la mayor parte de las veces, la instrucción PRINT se utiliza en modo indirecto, formando parte de un programa. En este caso, y tal como puede observarse en el programa que sigue, la ejecución tendrá lugar al comunicarle al ordenador la orden RUN:

```
10 PRINT
20 PRINT "CURSO"
30 PRINT "DE PROGRAMACION"
40 END
RUN

CURSO
DE PROGRAMACION
■
```

Cuando el dato a imprimir es un número, se omiten las comillas. Por ejemplo:

```
10 PRINT 15
20 PRINT 356.2
```

Lo mismo ocurre cuando el argumento de la instrucción PRINT está constituido por nombres de variables, ya sean numéricas (A, B, C...) o de cadena de caracteres (A$, B$, C$...). En tal caso, la instrucción PRINT imprimirá los valores que estén asignados a las variables en cuestión:

```
10 LET A$="EL NUMERO ES:"
20 LET N=8
30 PRINT A$;N
```

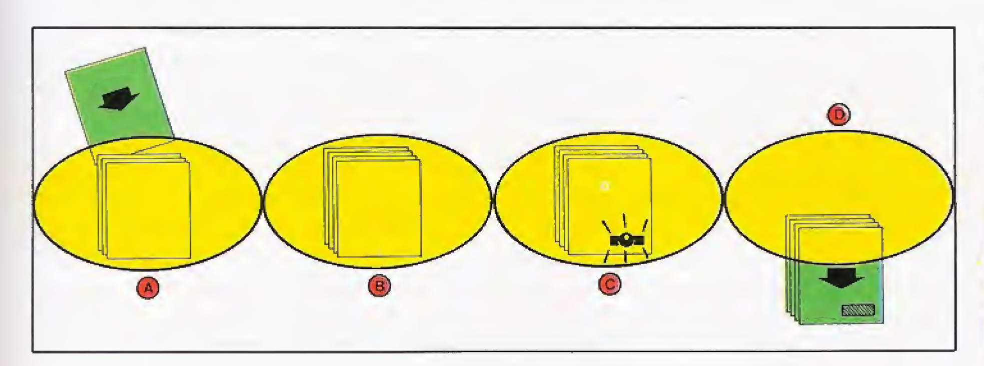
Las anteriores instrucciones constituyen un verdadero programa, al que sólo hay que añadir la instrucción final END.

TABLA DE CONVERSION							
ORDENADOR	**PRINT**						
	PRINT	**PRINT,**	**PRINT;**	**PRINT"<Mens.>"**	**PRINT X**	**PRINT<exp.>**	**PRINT<carácter de control>**
AMSTRAD	PRINT	PRINT,	PRINT;	PRINT"<Mens.>"	PRINT X	PRINT<exp.>	
APPLE II	PRINT	PRINT,	PRINT;	PRINT"<Mens.>"	PRINT X	PRINT<exp.>	
APRICOT	PRINT	PRINT,	PRINT;	PRINT' <Mens.>"	PRINT X	PRINT<exp.>	
ATARI	PRINT	PRINT,	PRINT;	PRINT"<Mens.>"	PRINT X	PRINT<exp.>	PRINT<carácter de control>
CBM 64	PRINT	PRINT,	PRINT;	PRINT"<Mens.>"	PRINT X	PRINT<exp.>	PRINT<carácter de control>
DRAGON	PRINT	PRINT,	PRINT;	PRINT"<Mens.>"	PRINT X	PRINT<exp.>	
EQUIPOS MSX	PRINT	PRINT,	PRINT;	PRINT"<Mens.>"	PRINT X	PRINT<exp.>	
HP-150	PRINT	PRINT,	PRINT[;]	PRINT"<Mens.>"	PRINT X	PRINT<exp.>	
IBM PC	PRINT	PRINT,	PRINT;	PRINT"<Mens.>"	PRINT X	PRINT<exp.>	
MPF	PRINT	PRINT,	PRINT;	PRINT"<Mens.>"	PRINT X	PRINT<exp.>	
NCR DN-V	PRINT	PRINT,	PRINT;	PRINT"<Mens.>"	PRINT X	PRINT<exp.>	
NEW BRAIN	PRINT	PRINT,	PRINT;	PRINT"<Mens.>"	PRINT X	PRINT<exp.>	
ORIC	PRINT	PRINT,	PRINT;	PRINT"<Mens.>"	PRINT X	PRINT<exp.>	
QL	PRINT	PRINT,	PRINT;	PRINT"<Mens.>"	PRINT X	PRINT<exp.>	
SHARP MZ-700	PRINT	PRINT,	PRINT;	PRINT"<Mens.>"	PRINT X	PRINT<exp.>	PRINT<carácter de control>
SPECTRAVIDEO	PRINT	PRINT,	PRINT;	PRINT"<Mens.>"	PRINT X	PRINT<exp.>	
SPECTRUM	PRINT	PRINT,	PRINT;	PRINT"<Mens.>"	PRINT X	PRINT<exp.>	

"<Mens.>": Mensaje o texto, entre comillas. <Exp.>: Expresión con dato o datos numéricos o alfanuméricos.

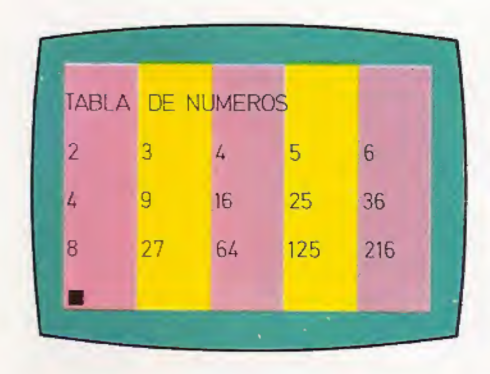
FORMULACIONES DEL COMANDO PRINT

PRINT: Salto a la próxima línea dejando una en blanco. PRINT,: Coloca el cursor unos espacios más adelante. PRINT;: Deja el cursor en el punto de impresión, sin desplazarlo. PRINT"<mensajes>": Escritura en pantalla del mensaje encerrado entre comillas. PRINT X: Escritura en pantalla de valor asignado a la variable X. PRINT (expresión): Imprime el valor de la expresión. PRINT<carácter de control>: Ejecuta la acción ordenada por el carácter de control.



Su ejecución, ordenada por medio del comando RUN, ilustra cuál es el efecto de los nombres de variables dispuestos en el argumento PRINT:

Las instrucciones indirectas se integran, precedidas de un número de línea, dentro de un programa (A y B), que sólo será ejecutado por el ordenador (C y D) al recibir la orden RUN.




```
10 LET A$="EL NUMERO ES"
20 LET N=8
30 PRINT A$
40 PRINT N
50 END
RUN
EL NUMERO ES:
8
■
```

En efecto, el ordenador ha llevado a la pantalla el contenido en ambas variables: una cadena de caracteres en el caso de A$ y un valor numérico en el de la variable N.

No terminan aquí las posibilidades de esta versátil instrucción. El argumento de PRINT puede también estar constituido por una expresión alfanumérica o matemática; ésta se verá resuelta por el ordenador antes de proceder a la presentación del resultado en pantalla.

```
10 LET=5
20 PRINT "SUMA"
30 PRINT 10+20
40 PRINT 15-A+2
50 END
RUN
SUMA
30
12
■
```

Separadores en el argumento de PRINT

Una misma instrucción PRINT puede utilizarse para trasladar a la pantalla di-

```
10 PRINT "TABLA DE NUMEROS"
20 PRINT 2,3,4,5,6
30 PRINT 4,9,16,25,36
40 PRINT 8,27,64,125,216
50 END
```

Tal como se observa, al emplear la coma (,) como separador de los datos en una sentencia PRINT, la pantalla se divide en un determinado número de campos o zonas de impresión (el número depende del intérprete BASIC).

versos datos. Estos deben incluirse dentro de la zona de argumento, convenientemente separados. Habitualmente, los signos que admite cualquier intérprete BASIC para separar a los distintos datos que acompañan al comando PRINT son la coma (,) y el punto y coma (;).

Al utilizar la coma como signo de separación, los datos aparecerán en la pantalla distribuidos en campos de una determinada longitud. Por el contrario, cuando el separador es el punto y coma, los datos se imprimirán uno inmediatamente a continuación de otro. Por ejemplo:

```
10 PRINT "DOS:",2
20 PRINT "TRES";3
30 END
RUN

DOS:        2
TRES:3
■
```

Al encontrar la coma, el ordenador escribirá ambos datos dejando un determinado espacio en blanco; éste será mayor o menor dependiendo de las características de cada intérprete BASIC. La distribución de campos o zonas de impresión dentro de una misma línea, originada por la presencia de este signo separador, resulta muy útil a la hora de construir tablas. Hay que tener en cuenta al respecto, que la distribución de campos o zonas de impresión dentro de una misma línea, originada por la presencia de este signo separador, resulta muy útil a la hora de construir tablas. Hay que tener en cuenta al respecto, que la distribución de los campos se mantiene en todas las líneas de la pantalla.

Los mencionados separadores («coma» y «punto y coma») también pueden utilizarse, con el mismo efecto, al final de la instrucción PRINT. En ambos casos, el cursor que señala el punto de impresión pasará a ocupar la posición que corresponda al separador utilizando antes de imprimir el argumento de la próxima instrucción PRINT. Veamos un programa claramente ilustrativo de ambas posibilidades:

```
10 PRINT "NUMEROS"
15 PRINT
20 PRINT 1,2,
30 PRINT 3
40 PRINT 4;5;6;
50 PRINT 7;8;9
60 END
```

Y éste es el resultado de su ejecución:



En efecto, la presencia de la coma al final de la instrucción 20 desactiva el salto a la próxima línea de pantalla, de tal forma que el dato 3, aportado por la siguiente instrucción PRINT, aparece en el próximo campo de la misma línea.

El punto y coma que cierra la instrucción 40 también inhibe el salto a la próxima línea, si bien, en este caso, el próximo dato (7) se visualizará adosado al último dato que figura en el argumento de la instrucción 40.

La instrucción 15 del ejemplo anterior ilustra otra de las peculiaridades del PRINT. Una instrucción PRINT puede estar constituida exclusivamente por el comando, desprovisto de argumento alguno. Su ejecución hará que aparezca en la pantalla una línea en blanco, pasando el cursor a ocupar la próxima línea de impresión.

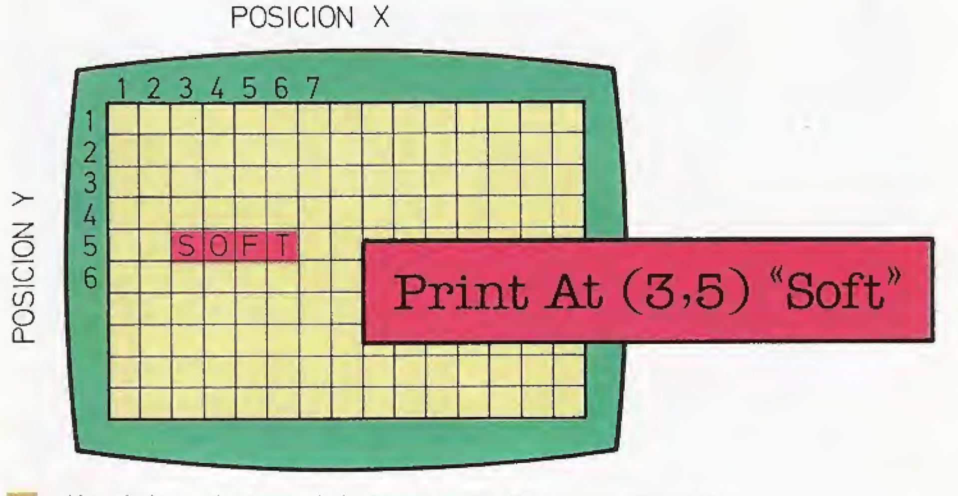
Educando a la máquina

Ejecución de programas y recolección de datos



Un primer contacto con la realidad del lenguaje BASIC lo proporciona el uso del comando PRINT. En sus distintas formulaciones, este comando permite trasladar información al órgano que refleja, a ojos del usuario, la actividad del ordenador.

Las posibilidades del comando PRINT no se limitan a la creación de los tipos de instrucciones cuyos formatos se analizaron en el capítulo precedente. Existen otras variantes, frecuentes en muchos intérpretes BASIC, que amplían el abanico de posibilidades del PRINT.



Una de las variantes de la instrucción PRINT es la que incorpora la función AT(X,Y). Con ella se puede seleccionar el punto de impresión del texto que figura en el argumento.

Variantes de la instrucción PRINT

Por el momento sólo se han utilizado instrucciones PRINT formuladas de acuerdo a su formato básico, cuya expresión general es:

(NL) PRINT <*expresión 1*>{[;] [,]}<*expresión 2*>...

Al utilizar este formato básico como instrucción indirecta, hay que incluir el número de línea (NL). Este precede al comando PRINT y a su argumento que, cabe recordar, puede estar constituido por uno o varios datos o expresiones.

Ciertos intérpretes BASIC, admiten otras variantes en la formulación del comando PRINT además de la que se ha estudiado como caso general.

Una de ellas es la que obedece al siguiente formato:

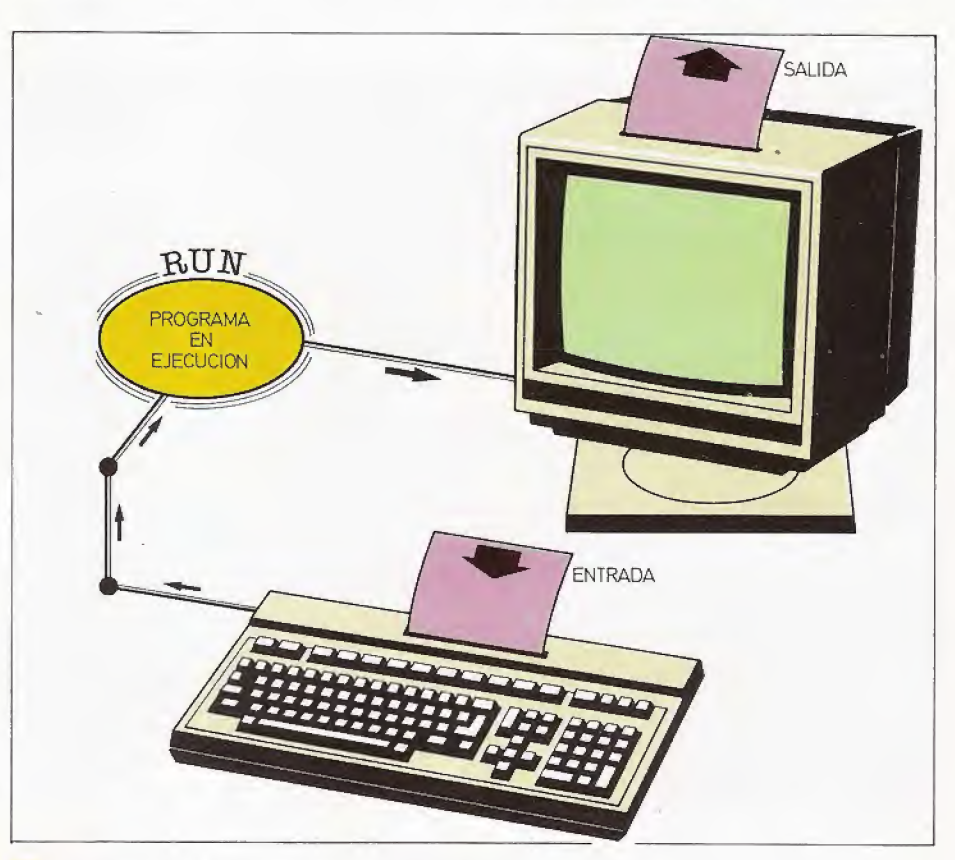
PRINT AT(x,y) <argumento>

La función AT(X,Y) que sigue al comando PRINT permite al usuario precisar el punto de la pantalla en que desea visualizar el argumento. Para ello, debe especificar los valores de X (columna: coordenada (horizontal) y de Y (fila: coordenada vertical), teniendo en cuenta que el origen de coordenadas se encuentra en el ángulo superior izquierdo de la pantalla (ver figura adjunta). Por ejemplo, la instrucción siguiente:

PRINT AT(5,2) "JUAN"

escribirá la palabra JUAN en la segunda fila de texto de la pantalla y a partir de la quinta columna; o lo que es lo mismo, dejando cuatro espacios en blanco a partir del margen izquierdo de la pantalla.

Otra de las variantes de la instrucción PRINT resulta especialmente adecuada para escribir en la pantalla dejando un determinado número de espacios en blanco; algo semejante a lo que permi-



El destino de todo programa es su ejecución en el ordenador. La orden BASIC al efecto es RUN.

RUN

Orden para la ejecución del programa en curso, almacenado en memoria.

Formato: RUN *<número de línea>* <, R>

Ejemplos: RUN
RUN 26
RUN 100,R.

ten las tabulaciones de una máquina de escribir. Su formato genérico es:

(NL) PRINT TAB(N); *<argumento>*

Su utilidad es manifiesta a la hora de confeccionar tablas, puesto que dando valores fijos a N, pueden seleccionarse perfectamente las columnas en las que se realizará la presentación de los datos. Cabe indicar al respecto que el valor de N debe coincidir con el número de espacios en blanco que quieran dejarse desde el margen izquierdo de la pantalla hasta el punto de escritura del primer carácter del argumento.

Existe una tercera variante también destinada a precisar el punto de escritura sobre la pantalla. Su formato es:

(NL) PRINT SPC(N); *<expresión 1>*; SPC(M); *<expresión 2>*...

El dato o valor de la expresión que sigue a cada función SPC se escribirá tantos espacios a la izquierda de la última posición escrita como dicte el parámetro que acompaña a SPC (N, M...).



El ordenador es una máquina a la que es preciso instruir hasta en el más mínimo detalle. Si no se le ayuda con la instrucción END no es capaz de encontrar el final del programa.

RUN

Por último, hay que constatar que determinados dialectos BASIC, permiten la sustitución de la palabra comando PRINT por el símbolo de cierre de interrogación (?). No existe diferencia alguna en el comportamiento y, realmente, la única justificación se encuentra en el intento de hacer más cómoda y rápida la escritura de las instrucciones PRINT. Por ejemplo, las dos siguientes líneas de programa son coincidentes desde el punto de vista de su ejecución:

```
20 PRINT "ALTERNATIVA AL COMANDO PRINT"
20 ? "ALTERNATIVA AL COMANDO PRINT"
```

A lo largo de la obra habrá ocasión de comprobar la utilidad práctica de todas las posibles versiones de la instrucción PRINT dentro de los programas BASIC.

Ejecución del programa

El destino de cualquier programa no es otro que su ejecución en el ordenador. Para cursar esta orden a la máquina, existe un comando BASIC al efecto: RUN.

Dada su naturaleza de comando de control, RUN se utiliza a modo de instrucción directa, sin número de línea. Puede introducirse en el ordenador en cualquier instante en que el intérprete BASIC esté dispuesto para recibir un mensaje.

Su ejecución provoca un borrado inicial de todas las variables en orden a que el programa no arrastre ninguna condición inicial que pueda entorpecer la ejecución y conducir a un resultado erróneo.

Al introducir el comando RUN desprovisto de argumento, la ejecución empezará a partir de la primera línea del pro-

grama. Si fuera necesario empezar la ejecución desde cualquier otra línea distinta de la inicial, habrá que especificar el número de línea en cuestión en la zona de argumento.

El formato genérico de una instrucción RUN es el que sigue:

RUN *<número de línea> <,R>*

Tal como veremos más adelante, con ocasión del estudio de los archivos de información en el BASIC, la opción final "R", permite mantener abiertos todos los ficheros de trabajo que ya se encontraban en esta situación antes de ejecutar el programa.

El comando END

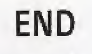
La necesidad de instruir al ordenador precisando cualquier matiz, por obvio que éste sea, es una realidad casi proverbial. Por si fuera preciso corroborarlo, aquí está la instrucción END, cuyo cometido no es otro que advertir al ordenador que ha llegado al final del programa. Por supuesto, no cabe argumento alguno para complementar la actuación de este comando, que por sí solo, constituye una instrucción completa.

Tras ejecutarla, el ordenador vuelve al «modo comando» (el cursor regresa a la pantalla) y el intérprete BASIC queda dispuesto para seguir prestando su eficaz servicio de intermediario con la interioridad del ordenador.

Los datos del BASIC

Sin lugar a dudas la misión primordial del ordenador es el tratamiento de los datos. Pero... ¿Qué tipos de datos? En esencia, un ordenador es una máquina de calcular, lo que significa que su mayor habilidad consiste en trabajar con números. De hecho, sus circuitos más elementales sólo reconocen códigos numéricos (cadenas de ceros y unos). No obstante, un ordenador dotado de un intérprete BASIC es ya capaz de identificar varios tipos de datos. En el BASIC cabe distinguir dos tipos de datos:

- *numéricos y*
- *alfanuméricos.*



Instrucción de fin de programa.

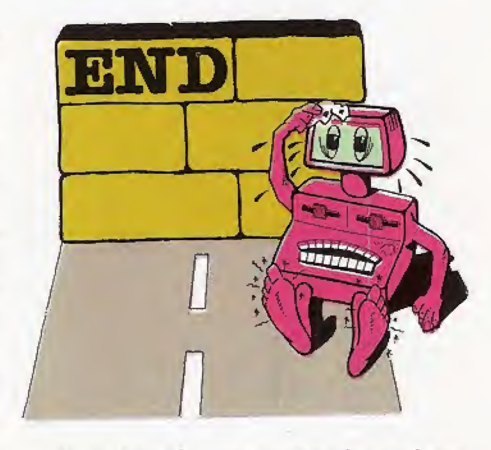
Formato: (Número de línea) END

Ejemplos: END

Los primeros son ni más ni menos que números convencionales expresados, normalmente, en el sistema decimal. Al segundo tipo pertenece cualquier cadena de caracteres o conjunto de letras, números y caracteres especiales (signos de puntuación, etc.).

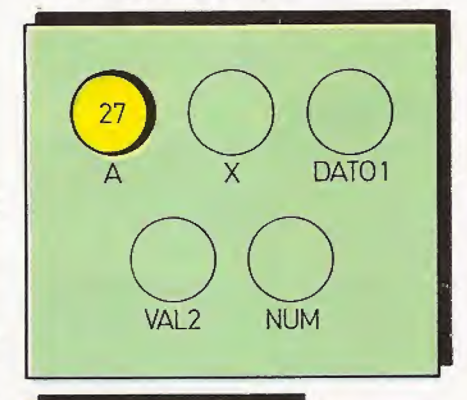
Según esta clasificación, un número puede ser considerado como dato numérico o alfabético. Para diferenciar su naturaleza, los números utilizados como datos alfanuméricos suelen ir encerrados entre comillas. Por ejemplo: 1 es un dato numérico, mientras que "1" corresponde a su expresión alfanumérica.

Ambos tipos de datos están presentes en cualquier programa BASIC. La evidencia la encontramos en algunos de los ejemplos propuestos al tratar el comando PRINT.



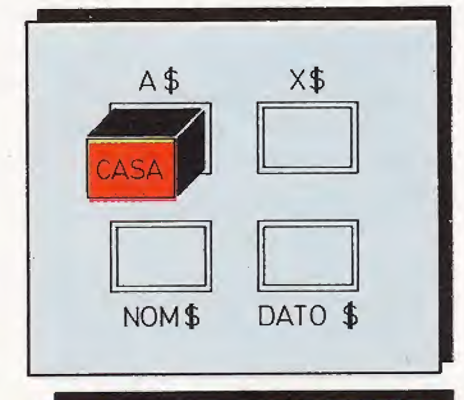
Al hablar de este comando, se introdujo el concepto de variable o referencia simbólica a la que pueden asignarse datos fijos o constantes. Las variables tienen en el BASIC una función pareci-

Variables Numéricas



LET A = 27

Variables Alfanumèricas



LET A$ = "CASA"

LET es el comando BASIC adecuado para construir las instrucciones de asignación. Su argumento contiene el nombre de la variable y el dato que hay que asignar a la misma; ambos elementos, relacionados por el signo «igual» (=), deben ser del mismo tipo (numéricos o alfanuméricos).

LET

Asignación de datos o expresiones a variables.

Formato: (Número de línea) LET *<nombre de variable>*=*<expresión>*

Ejemplos:
```
   LET A=7
10 LET AX$="VARIABLE"
20 LET SUMA=20+250+C
30 LET LONG=2*PI*R
```

```
10 LET R=5
20 LET PI=3.141592
30 LET LONG=2*PI*R
40 PRINT "LA LONGITUD ES: "; LONG
50 END
RUN

LA LONGITUD ES: 31.416
```

da a la propia de las variables matemáticas. Estas últimas se utilizan para designar a un dato desconocido o que puede tomar diferentes valores. En el lenguaje BASIC, las variables tienen un «nombre» que sirve para identificarlas, y un «contenido» o valor que toma la variable en un determinado momento.

Una de las operaciones más frecuentes dentro de un programa es, precisamente, la de alterar el contenido de algunas variables. Este proceso se realiza por medio de denominadas «sentencias de asignación», cuyo cometido es asignar a una variable su correspondiente valor. Este coincidirá con un dato de uno de los dos tipos comentados. Una vez realizada la asignación, el valor o contenido de una variable puede ser utilizado como un simple dato. El comando encargado de la asignación es LET. Su formato es el siguiente:

LET <nombre de variable>=<expresión>

El campo denominado <expresión> puede contener un número, una cadena de caracteres o, en general, una combinación de datos y operadores. Por supuesto, los datos pueden ser constantes o valores numéricos o alfanuméricos o, sencillamente, nombres de variables representativas de su contenido. Las siguientes son asignaciones válidas:

```
LET PI=3.141592
LET NOM$="PACO"
LET SUMA=5+3
LET LONG=2*PI*R
```

En el primer ejemplo se asigna a la variable PI el valor numérico 3.141592. Una vez definida una variable, ésta puede ya utilizarse como dato en otra asignación; la cuarta instrucción LET, utiliza la variable PI (definida en la primera línea) dentro de la expresión cuyo resultado se asigna a la variable LONG. Como se observa, para acceder al valor de una variable basta tan sólo con «llamarla» por su nombre.

El siguiente programa utiliza tres instrucciones de asignación:

```
10 LET R=5
20 LET PI=3.141592
30 LET LONG=2*PI*R
40 PRINT "LA LONGITUD ES: "; LONG
50 END
```

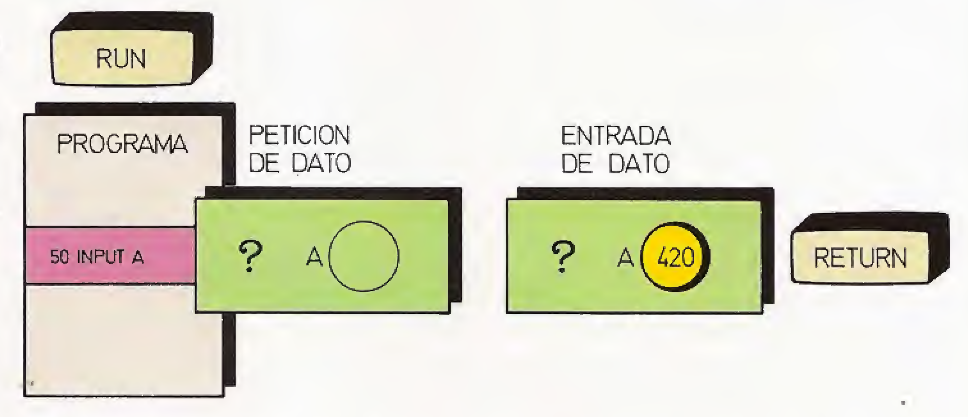
El ejemplo calcula la longitud de la circunferencia cuyo radio (R) se especifica en la línea 10. Para determinar la longitud de una circunferencia de distinto radio, basta con otorgar a R el nuevo valor dentro de la mencionada instrucción.

Cabe observar que los números aparecen en notación inglesa, sustituyendo a la coma decimal por un punto. Esta es una característica casi generalizada en los intérpretes BASIC.

En la mayor parte de los dialectos BASIC, la presencia de la palabra LET es opcional; puede omitirse el comando manteniendo, por supuesto, el formato característico de las instrucciones de asignación. El ejemplo anterior adoptará, en este caso, la forma:

```
10 R=5
20 PI=3.141592
30 LONG=2*PI*R
40 PRINT "LA LONGITUD ES: "; LONG
50 END
```

Ciertas versiones del BASIC admiten la posibilidad de realizar múltiples asignaciones dentro de una misma instrucción LET. El dato o expresión se asigna



Al ejecutar una instrucción INPUT, el ordenador detiene la secuencia de ejecución y solicita al usuario el dato que hay que asignar a la variable que constituye su argumento.

simultáneamente a todas las variables que lo preceden. Por ejemplo:

```
LET A=B=C=25
```

En esta ocasión, las variables A, B, C tomarán todas ellas el valor 25.

Recolectando datos

En este punto de la obra, el BASIC ha desvelado ya algunos comandos de su repertorio. Comandos útiles para realizar un determinado tratamiento de la información: PRINT (presentar datos en la pantalla), LET (asignar datos a variables)... Se han aportado, incluso, algunos programas sencillos pero ilustrativos de las posibilidades que brinda el BASIC para manipular datos.

Parece obvio que el próximo paso hay que darlo en el sendero de los datos; presentando uno de los comandos BASIC destinado a la captación de datos. INPUT es uno de los comandos de esta categoría. Su especialidad es la de gestionar la entrada de datos durante la ejecución del programa. Al encontrar una instrucción INPUT, el ordenador detiene la secuencia de ejecución y solicita al usuario los datos exigidos por la mencionada instrucción.

La captación de datos a través de INPUT, se reduce a un proceso de asignación. Veamos un ejemplo introductorio:

```
10 PRINT "¿COMO TE LLAMAS?"
20 INPUT A$
30 PRINT "HOLA"; A$
40 END
RUN

¿COMO TE LLAMAS?
?
```

La línea 20 revela uno de los formatos tradicionales de la instrucción INPUT: el comando, seguido de una variable. Como ya se ha mencionado, la introducción de datos se concreta en una asignación. En el ejemplo, el dato que se introduzca quedará asignado a la variable A$, que constituye el argumento; por supuesto, la naturaleza alfanumérica de la variable (A$) obliga a que el dato de entrada sea una cadena de caracteres.

Regresemos de nuevo al ejemplo. Al llegar a la instrucción INPUT, se detiene el proceso de ejecución y aparece un interrogante en la pantalla.

El signo de interrogación que precede al cursor indica, ni más ni menos, que el programa aguarda a que el usuario introduzca un dato. Para que la ejecución pueda continuar, es necesario introducir el dato solicitado, seguido por una acción sobre la tecla RETURN. El efecto de la orden RETURN no es otro que identificar el final del dato.

```
RUN

¿COMO TE LLAMAS?
?MANUEL(RT)

HOLA MANUEL
```

Tras recibir el dato en cuestión, el ordenador vuelve a ocuparse del programa: asigna a A$ el dato "MANUEL" y,

INPUT

Asigna el valor introducido por el teclado a la variable indicada.

Formato: (Número de línea INPUT [;] ["<*Mensaje*>" ;,] <*var. 1*>[,<*var. 2*>...]

Ejemplos: INPUT A$
INPUT "VALOR";V
INPUT "DIA",D

VARIANTES DE LA INSTRUCCION PRINT

Formato: (Número de línea) PRINT AT(X,Y) <*argumento*>

Ejemplos: 20 PRINT AT(20,12) "JUAN PEREZ"
40 PRINT AT(5,3) "NOMBRE:";A$

Definición: Escribe el argumento a partir del punto X,Y de la pantalla; siendo X el número de columna e Y el número de fila. El origen de coordenadas se encuentra en el ángulo superior izquierdo de la pantalla.

Formato: (Número de línea) PRINT TAB(N); <*argumento*>

Ejemplos: 45 PRINT TAB(5); "TABLA"
60 PRINT TAB(25); C$,D

Definición: Escribe el argumento a N espacios de distancia del margen izquierdo de la pantalla. Actúa de forma análoga al tabulador de una máquina de escribir.

Formato: (Número de línea) PRINT SPC(N); <*expresión 1*>: SPC(M); <*expresión 2*>...

Ejemplos: 30 PRINT SPC(7); "ARTICULO"; SPC(15) "PRECIO"
40 PRINT SPC(10); A$; SPC(18); P

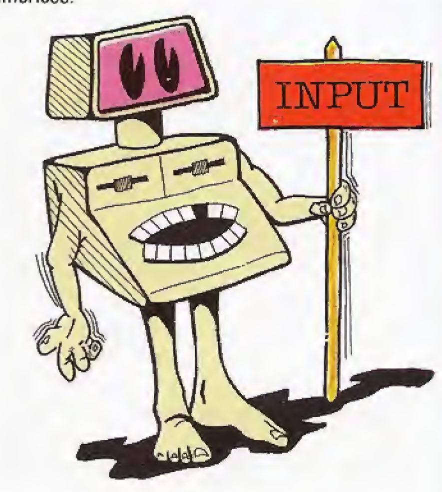
Definición: Imprime la expresión correspondiente a N (o M) espacios a la izquierda de la posición en la que se encontraba el cursor.

TABLA DE CONVERSION (1)					
Ordenador	RUN		END	LET	
	RUN	RUN nl	END	LET <var.>=<expr.>	<var.>=<expr.>
AMSTRAD	RUN	RUN nl	END	LET <var.>=<expr.>	<var.>=<expr.>
APPLE II (APPLESOFT)	RUN	RUN nl	END	LET <var.>=<expr.>	<var.>=<expr.>
APRICOT (M-BASIC)	RUN	RUN nl	END	LET <var.>=<expr.>	<var.>=<expr.>
ATARI	RUN	RUN nl	END	LET <var.>=<expr.>	<var.>=<expr.>
CBM 64	RUN	RUN nl	END	LET <var.>=<expr.>	<var.>=<expr.>
DRAGON	RUN	RUN nl	END	LET <var.>=<expr.>	<var.>=<expr.>
EQUIPOS MSX	RUN	RUN nl	END	LET <var.>=<expr.>	<var.>=<expr.>
HP-150	RUN	RUN nl	END	LET <var.>=<expr.>	<var.>=<expr.>
IBM PC	RUN	RUN nl	END	LET <var.>=<expr.>	<var.>=<expr.>
MPF	RUN	RUN nl	END	LET <var.>=<expr.>	<var.>=<expr.>
NCR DM-V (MS-BASIC)	RUN	RUN nl	END	LET <var.>=<expr.>	<var.>=<expr.>
NEW BRAIN	RUN	GOTO nl	END	LET <var.>=<expr.>	<var.>=<expr.>
ORIC	RUN	RUN nl	END	LET <var.>=<expr.>	<var.>=<expr.>
SHARP MZ-700 (MZ-BASIC)	RUN	RUN nl	END	LET <var.>=<expr.>	<var.>=<expr.>
SINCLAIR QL	RUN	RUN nl	—	LET <var.>=<expr.>	<var.>=<expr.>
SPECTRAVIDEO	RUN	RUN nl	END	LET <var.>=<expr.>	<var.>=<expr.>
ZX-SPECTRUM	RUN	RUN nl	—	LET <var.>=<expr.>	<var.>=<expr.>

nl: Número de línea. <var.>: Variable. <expr.>: Expresión. Dato o combinación de datos numéricos o alfanuméricos.

FORMULACIONES DE LOS COMANDOS

RUN: Ejecuta en su totalidad el programa que se encuentra en la memoria central. RUN nl: Ejecuta el programa a partir de la línea específica (nl). END: Señala el final del programa.



por medio de la instrucción 40, lo visualiza en la pantalla precedido del mensaje "HOLA".

Ya se ha mencionado el hecho de que la variable de solicitud de dato (A$ en el ejemplo) puede ser numérica o alfanumérica, exigiendo, en cada caso, un dato del tipo solicitado: número o cadena de caracteres.

Son muchas las posibilidades que este comando pone al alcance del usuario. Una de ellas deriva de la posibilidad de introducir un mensaje en el argumento de INPUT, de tal forma que la solicitud del dato incorpore un texto al efecto.

Veamos un ejemplo. Se trata, sencillamente, de un programa capaz de calcular el precio total de un determinado número de artículos del mismo tipo. El programa incluye las instrucciones INPUT necesarias para pedir al usuario la cantidad de artículos vendidos y el precio por unidad.

TABLA DE CONVERSION (2)

Ordenador	INPUT				
	INPUT <var.>	INPUT "<mens.>";<var.>	INPUT "<mens.>",<var.>	INPUT <var. 1>,<var. 2>...	INPUT;"<mens.>"...
AMSTRAD	INPUT <var.>	INPUT "<mens.>";<var.>	INPUT "<mens.>",<var.>	INPUT <var. 1>,<var. 2>...	INPUT;"<mens.>"...
APPLE II (APPLESOFT)	INPUT <var.>	—	—	INPUT <var. 1>,<var. 2 >...	—
APRICOT (M-BASIC)	INPUT <var.>	INPUT "<mens.>";<var.>	INPUT "<mens.>",<var.>	INPUT <var. 1>,<var. 2>...	INPUT;"<mens.>"...
ATARI	INPUT <var.>	—	—	INPUT <var. 1>,<var. 2>...	—
CBM 64	INPUT <var.>	INPUT "<mens.>";<var.>	—	INPUT <var. 1>,<var. 2>...	—
DRAGON	INPUT <var.>	INPUT "<mens.>";<var.>	—	INPUT <var. 1>,<var. 2>...	—
EQUIPOS MSX	INPUT <var.>	INPUT "<mens.>";<var.>	INPUT "<mens.>",<var.>	INPUT <var. 1>,<var. 2>...	INPUT;"<mens.>"...
HP-150	INPUT <var.>	INPUT "<mens.>";<var.>	INPUT "<mens.>",<var.>	INPUT <var. 1>,<var. 2>...	INPUT;"<mens.>"...
IBM PC	INPUT <var.>	INPUT "<mens.>";<var.>	INPUT "<mens.>",<var.>	INPUT <var. 1>,<var. 2>...	INPUT;"<mens.>"...
MPF	INPUT <var.>	—	INPUT "<mens.>",<var.>	INPUT <var. 1>, <var. 2>...	—
NCR DM-V (MS-BASIC)	INPUT <var.>	INPUT "<mens.>";<var.>	INPUT "<mens.>",<var.>	INPUT <var. 1>,<var. 2>...	INPUT;"<mens.>"...
NEW BRAIN	INPUT <var.>	—	INPUT ("<mens.>") <var.>	INPUT <var. 1>,<var. 2>...	—
ORIC	INPUT <var.>	INPUT "<mens.>";<var.>	—	INPUT <var. 1>,<var. 2>...	—
SHARP MZ-700 (MZ-BASIC)	INPUT <var.>	INPUT "<mens.>";<var.>	—	INPUT <var. 1>,<var. 2>...	—
SINCLAIR QL	INPUT <var.>	INPUT "<mens.>";<var.>	—	INPUT <var. 1>,<var. 2>...	—
SPECTRAVIDEO	INPUT <var.>	INPUT "<mens.>";<var.>	—	INPUT <var. 1>,<var. 2>...	—
ZX-SPECTRUM	INPUT <var.>	INPUT "<mens.>";<var.>	—	INPUT <var. 1>,<var. 2>...	—

<var.>: Variable. <mens.>: Texto o mensaje.

FORMULACIONES DEL COMANDO INPUT

INPUT <var.>: Entrada de un dato asignándolo a la variable indicada. INPUT "<mens.>";<var.>: Entrada de un dato con presentación del mensaje en la pantalla seguido por el signo de interrogación. INPUT "<mens.>",<var.>: Entrada de un dato con presentación del mensaje omitiendo el signo de interrogación. INPUT <var. 1>,<var. 2>...: Entrada de un conjunto de datos, separados por comas, asignándolas a las variables en el orden en el que éstas aparecen. INPUT;"<mens.>"...: Entrada de datos sin que se produzca un salto de línea tras su introducción.

```
10 INPUT "CANTIDAD: "; C
20 INPUT "PRECIO UNITARIO": "; PT
30 PRINT "IMPORTE TOTAL: "; C*PT
40 END
RUN

CANTIDAD: ?5(RT)
PRECIO UNITARIO: ?30 (RT)
IMPORTE TOTAL: 150
```

Otra de las virtudes de INPUT es que tras la introducción de un dato erróneo, no «rompe» la ejecución del programa. En su lugar muestra un *mensaje de error* y, a continuación, vuelve a solicitar el dato.

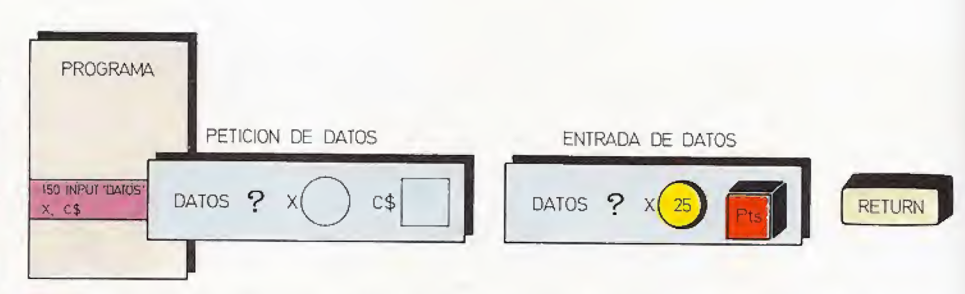
El signo separador entre el mensaje y la variable que recoge el dato introducido puede ser una coma en lugar de un punto y coma. Esta alternativa suprime la interrogación.

```
10 INPUT "INTRODUZCA SU NOMBRE: ", A$
20 PRINT "HOLA"; A$
30 END
RUN

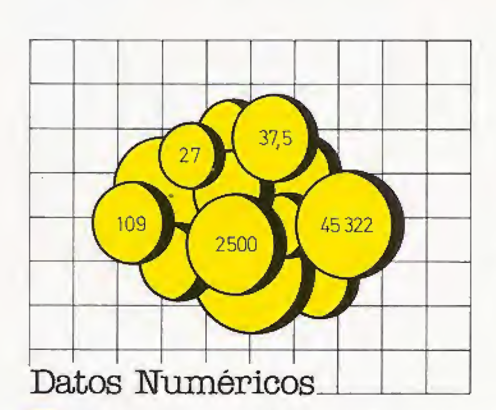
INTRODUZCA SU NOMBRE: MIGUEL (RT)
HOLA MIGUEL
```

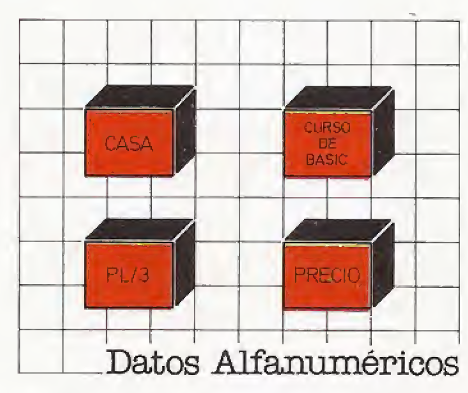
En esta nueva versión del ejemplo inicial, se observa que al utilizar la coma como elemento separador desaparece el signo de interrogación que visualiza el BASIC.

Otra particularidad adicional. Si el comando INPUT inmediatamente va seguido por un punto y coma, la orden RETURN que pone fin al dato introducido no provocará un salto a la siguiente línea de impresión. El cursor permanecerá junto al dato ingresado por el usuario. Ello supone que el próximo mensaje se visualizará en la misma línea. Para observar el resultado basta con modificar la línea 10 del ejemplo anterior.



Una misma instrucción INPUT es utilizable para la captación de varios datos, incluso de distinto tipo. Tras introducir los datos hay que accionar la tecla RETURN para que prosiga la ejecución del programa.





En el lenguaje BASIC coexisten dos tipos de datos: numéricos (números convencionales expresados, normalmente, en el sistema decimal) y alfanuméricos o cadenas de caracteres (conjuntos de letras, números y caracteres especiales).

```
10 INPUT; "INTRODUZCA SU NOMBRE: ", A$
```

El resultado de la nueva ejecución será:

```
RUN

INTRODUZCA SU NOMBRE PABLO: HOLA PABLO
```

Por último, cabe añadir que una sola instrucción INPUT es perfectamente utilizable para la captación de varios datos, e incluso de distinto tipo. Los datos introducidos se irán asignando, ordenadamente, a las variables incluidas en el argumento. El número de datos introducidos debe ser igual al número de variables que aparecen en la lista y, por supuesto, su naturaleza (dato numérico o alfanumérico) debe coincidir con la de la variable correspondiente. Por ejemplo:

```
INPUT "TRES DATOS"; A,B,C
INPUT "FECHA(AÑO,MES,DIA)";
A,MES$,DIA
```

En el primer caso, hay que responder con tres datos numéricos. No obstante, como respuesta a la segunda instrucción INPUT será preciso introducir un número (A:año), seguido por una cadena de caracteres (MES$), para terminar con un nuevo valor numérico (DIA). Tanto en este caso como en cualquier otra situación análoga, en la que haya que introducir varios datos como respuesta a un INPUT, éstos se separarán por medio de una coma.

La versatilidad que permite el comando INPUT puede llegar a simplificar las tareas de programación de forma más que apreciable. Es al usuario a quien corresponde optar por la formulación idónea en cada caso.

Un simple programa, capaz de pedir la introducción de una serie de seis números, operar la suma y presentar el resultado en la pantalla, puede ilustrar la importancia que adquiere la elección del método.

```
20 LET S=0
30 INPUT "A";A
40 LET S=S+A
50 INPUT "B";B
60 LET S=S+B
70 INPUT "C";C
80 LET S=S+C
90 INPUT "D";D
100 LET S=S+D
110 INPUT "E";E
120 LET S=S+E
130 INPUT "F";F
140 LET S=S+F
150 PRINT "SUMA TOTAL=";S
160 END
```

Es evidente que el procedimiento elegido no es el más adecuado. El comando INPUT admite otras formulaciones capaces de reducir la longitud del programa y hacer más cómoda la introducción de los datos. Por ejemplo:

```
20 INPUT "A,B,C,D,E,F";A,B,C,D,E,F
30 LET S=A+B+C+D+E+F
40 PRINT "LA SUMA ES: "; S
50 END
```

Ambos programas son de todo punto equivalentes; introduciendo los mismos valores en ambos casos, el resultado será el mismo. Sin embargo, no cabe duda que el procedimiento correcto es el utilizado en el segundo programa.

Operando con el BASIC

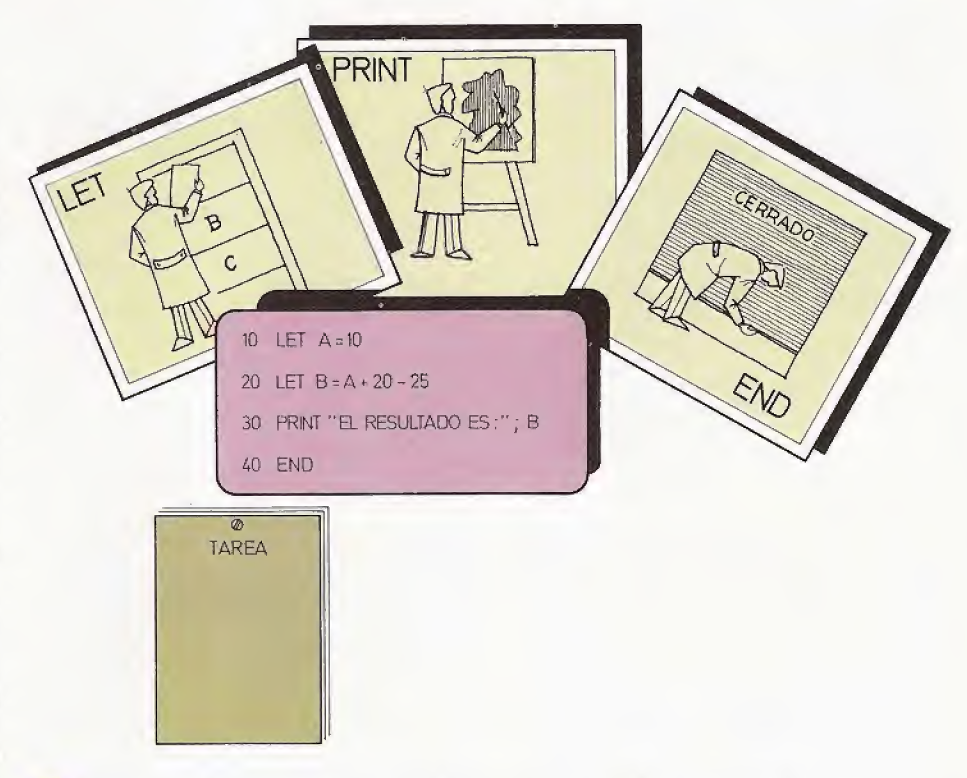
Listados y comentarios. Operadores aritméticos fundamentales



Los ordenadores son, en esencia, máquinas concebidas para almacenar y procesar información. El concepto de información relativo a los ordenadores tiene algo que ver con la aceptación coloquial de este término.

En general, información es todo aquello que incrementa nuestro conocimiento. Trasladándolo al caso del ordenador, el concepto de información es aplicable a todo aquel material que se suministra a la máquina, ya sea para instruirla (comandos, órdenes, instrucciones, programas) o para que ésta lo opere y manipule de acuerdo a las indicaciones que reciba (datos).

La confección de programas —información destinada a «instruir a la máquina»— es, precisamente, el objetivo práctico de los lenguajes de programación. Estos deben aportar el vocabulario adecuado para expresar cualquiera de las acciones habituales; además, deben ofrecer al usuario un surtido de órdenes que faciliten la confección de programas. Dentro de este último grupo se encuentran dos comandos BASIC que se van a exponer a continuación: LIST y REM.



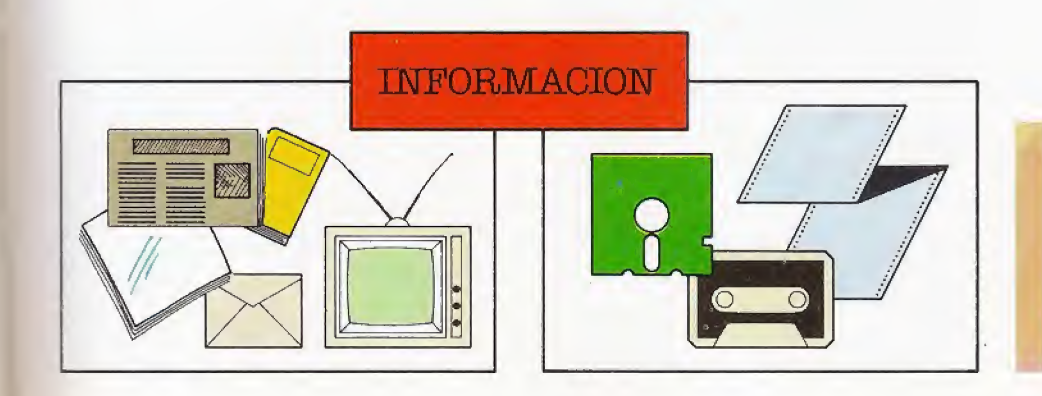
El objetivo de los lenguajes de programación es construir programas: secuencias ordenadas de instrucciones capaces de «educar» al ordenador para que realice un determinado trabajo.

El comando LIST

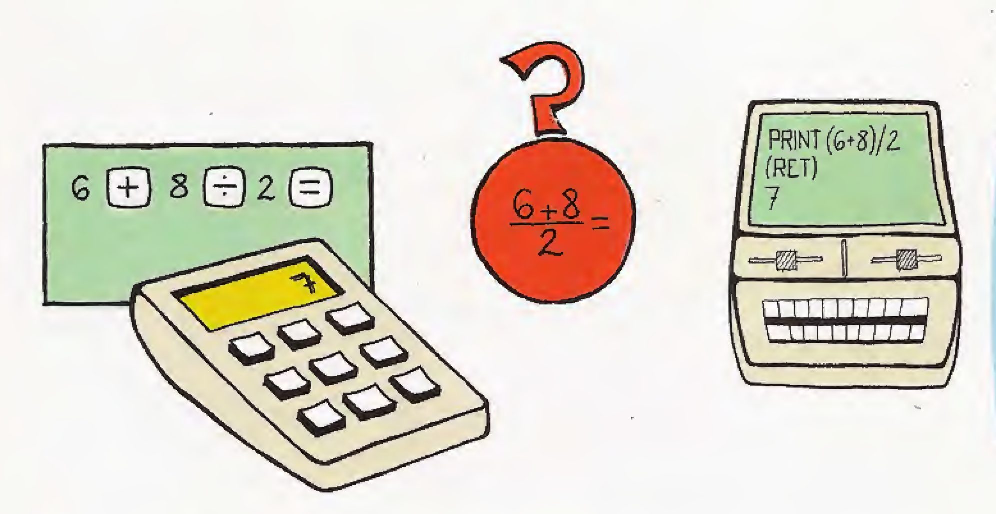
Una necesidad obvia del programador, es la de ver en cualquier momento el resultado de su trabajo. En definitiva, obtener en la pantalla una lista ordenada de las instrucciones que ha ingresado en el ordenador para construir un programa.

Tal posibilidad la ofrece el comando LIST. Este permite *visualizar* (LISTar) el programa almacenado en memoria en ese preciso instante.

Pero no radica ahí la solución de todo el problema... ¿Qué sucede cuando el programa es suficientemente grande como para no poder visualizarlo en el espacio que ofrece una sola pantalla? Para solucionar este inconveniente, el comando LIST ofrece toda una serie de opciones que permiten el cómodo examen del programa a través de la pantalla.



En esencia, los ordenadores son máquinas concebidas para almacenar y procesar información. En el caso del ordenador, el concepto de información es aplicable a todo aquello que se suministra a la máquina, ya sea para instruirla (órdenes, instrucciones, programas) o para que proceda a su tratamiento (datos).



El ordenador dotado de un intérprete de lenguaje BASIC puede utilizarse a modo de calculadora. Para ello hay que recurrir al empleo de instrucciones formuladas en modo directo. Por supuesto, hay que respetar el modo de introducción adecuado.

Las instrucciones LIST, cuyo formato general es:

LIST [<Número de línea> [—[<Número de línea>]]]

permiten «listar», total o parcialmente, el programa que se encuentra almacenado en la memoria del ordenador.

Las distintas opciones que ofrece el comando, pueden ser seleccionadas mediante la inclusión o no de los *campos* opcionales a los que hace referencia el formato general.

Para estudiar las distintas modalidades de instrucciones que pueden construirse con el comando LIST, se utilizará un mismo programa ejemplo, cuyo listado completo es el que aparece a continuación:

```
10 PRINT "INSTRUCCIONES"
20 PRINT "LIST:"
30 PRINT "ESTO NO ES";
40 PRINT "MAS QUE UN"
50 PRINT "EJEMPLO"
60 END
```

La primera de las instrucciones a la que da pie el uso de este comando consta, sencillamente, del comando LIST aislado, omitiendo la zona de argumento. La respuesta del ordenador será la presentación del programa completo, listado a partir del número de línea inferior.

En el caso de que el programa que se encuentra en la memoria sea el propuesto en el párrafo anterior, el efecto de la orden LIST será el siguiente:

```
LIST

10 PRINT "INSTRUCCIONES"
20 PRINT "LIST:"
30 PRINT "ESTO NO ES";
40 PRINT "MAS QUE UN"
50 PRINT "EJEMPLO"
60 END
█
```

Como se observa, el ordenador muestra todas y cada una de las líneas del programa.

También es posible «listar» tan sólo parte del programa. Para ello, hay que especificar en el argumento de LIST la línea o las líneas deseadas.

Una primera alternativa es introducir como argumento un solo número de línea. La respuesta de la máquina será la visualización exclusiva de tal línea; por ejemplo:

```
LIST 40

40 PRINT "MAS QUE UN"
█
```

Si el número de línea va seguido por un guión, se listará la mencionada línea y todas las restantes que tengan un número superior al indicado. Estos es: se mostrará en la pantalla la zona del programa que va desde la línea tras el comando LIST hasta el final del mismo:

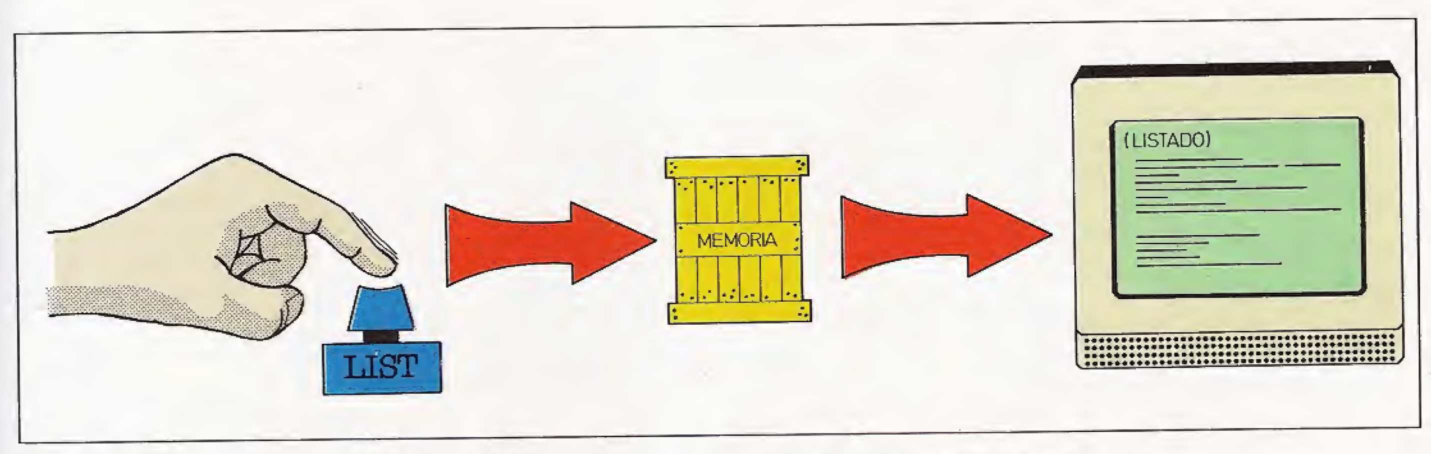
```
LIST 30—

30 PRINT "ESTO NO ES";
40 PRINT "MAS QUE UN"
50 PRINT "EJEMPLO"
60 END
█
```

Por el contrario, si el guión precede al número de líneas especificado, se listará el programa desde el principio hasta llegar a ese número de línea. Por ejemplo:

```
LIST —30

10 PRINT "INSTRUCCIONES"
20 PRINT "LIST:"
30 PRINT "ESTO NO ES";
█
```



La función del comando LIST, en sus distintas formulaciones, es presentar en la pantalla un listado total o parcial del programa almacenado en la memoria del ordenador.

Una nueva variante de la instrucción LIST es la que incluye en el argumento dos números de línea, separados por un guión. En tal caso, se listarán todas las líneas del programa cuyo número esté comprendido entre ambas. Esta última opción resulta útil par apresentar en pantalla sólo determinados bloques o zonas del programa en curso.

```
LIST 20–40

20 PRINT "LIST:"
30 PRINT "ESTO NO ES";
40 PRINT "MAS QUE UN"
█
```

Tras la ejecución del comando LIST, el BASIC regresa al modo directo. Ello significa que si se introduce tal comando en forma de instrucción indirecta dentro de un programa, la *ejecución* del mismo se detendrá tras listarlo. Un ejemplo de la actuación del comando LIST utilizado a modo de instrucción indirecta, lo aporta el siguiente programa:

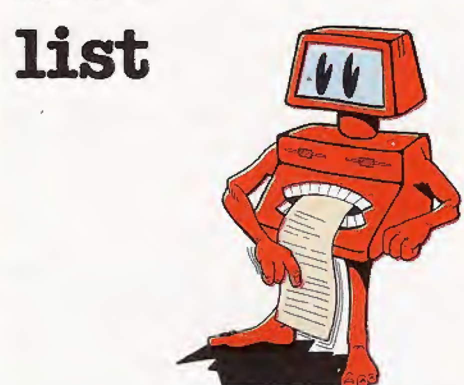
```
10 PRINT "LIMITACIONES DEL LIST"
20 PRINT "ESTA LINEA SI SE EJECUTA"
30 LIST
40 PRINT "ESTA LINEA NUNCA SE EJECUTARA"
50 END
```

En efecto, la ejecución se detiene al procesar la instrucción LIST: el ordenador presenta el listado del programa y, acto seguido, abandona la secuencia de ejecución para pasar a modo directo o situación de diálogo:

```
RUN
LIMITACIONES DEL LIST
ESTA LINEA SI SE EJECUTA
10 PRINT "LIMITACIONES DEL LIST"
20 PRINT "ESTA LINEA SI SE EJECUTA"
30 LIST
40 PRINT "ESTA LINEA NUNCA SE EJECUTARA"
50 END
█
```

Al recibir la orden RUN, el ordenador ha iniciado la ejecución del programa, cursando la tarea ordenada en las instrucciones 10 y 20. Al llegar a la línea 30, la máquina ejecuta la instrucción LIST, mostrando en pantalla el listado completo del programa... y, acto seguido, se detiene mostrando el cursor. Las líneas 40 y 50 quedan sin ejecutar, debido a que la instrucción LIST (línea 30) obliga al intérprete BASIC a abandonar la ejecución en curso y a regresar a modo directo.

list



LIST

Presenta en pantalla las líneas del programa almacenado en memoria.

Formato: LIST[<*número de línea*>[–[<*número de línea*>]]]

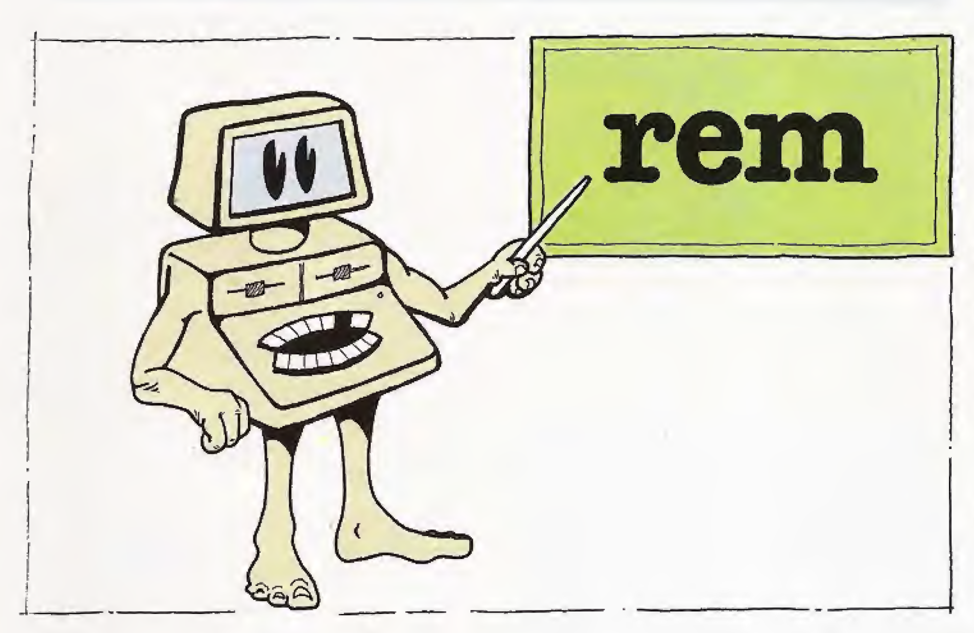
Ejemplos: LIST
LIST 3–100
LIST–200
LIST 50–

REM

Introduce un comentario en el programa. Este comando y el texto que le sigue no se ejecutan.

Formato: (Número de línea) REM[<comentario>]

Ejemplos: REM
REM ESTO ES UN COMENTARIO
REM PROGRAMA 56-B



El comando REM

A primera vista, la presencia de ese comando dentro del vocabulario de un lenguaje de programación, puede parecer un despropósito. REM es un comando que no tiene efecto alguno en la ejecución de un programa BASIC, hasta el punto de que es ignorado por el ordenador. No obstante, en determinados casos, su presencia dentro de un programa llega a ser casi esencial. La misión del REM (del inglés: REMARK, Comentario) es introducir comentarios en el programa que faciliten su interpretación posterior por parte del propio programador o de otros usuarios. En programas cortos su empleo es casi superfluo, ya que con un simple vistazo se puede saber cuál es el cometido del programa y la función de las variables empleadas.

Operadores ariméticos básicos			
Operación	**Comentario**	**Ejemplo**	**Resultado**
+	suma	6+4	10
–	resta	3-8	-5
*	multiplicación	5*7	35
/	división	8/4	2
^ ó ↑	potención	6↑3	216

No obstante, cuando el programa sea un poco largo y complicado, su presencia resultará providencial. Si no se introducen comentarios explicando la función de cada parte del programa y el cometido de las distintas variables, ni al mismo programador que lo ha diseñado le resultará fácil revisar su estructura e introducir nuevas modificaciones en tal programa, transcurrido un cierto tiempo desde su confección.

```
10 REM PRINCIPIO DE LA ZONA DE CALCULO
20 REM PROGRAMA REVISADO EL 10 DE OCTUBRE
30 REM LA VARIABLE P CONTIENE EL PRECIO EN DOLA-
   RES
```

Como se observa en el ejemplo, el formato del comando REM es de lo más simple. Basta con empezar la instrucción con la palabra REM y añadir a continuación el texto del comentario.

En algunas versiones del lenguaje BASIC, es posible sustituir la palabra clave REM, por un simple asterisco (*), por un apóstrofe ('), por un signo de admiración (!) o por cualquier otro símbolo específico.

Operadores aritméticos

Sin lugar a dudas, un ordenador desde luego es mucho más potente que una calculadora. En principio, puede compartir todas las posibilidades de una calculadora: desde las operaciones aritméticas elementales hasta cálculos en los que intervengan funciones trigonométricas y algebraicas.

A pesar de ello, hay que señalar que el modo de operación no acostumbra a ser tan inmediato como el propio de una calculadora. No hay que perder de vista que, en el caso del ordenador, hay que ajustarse a las reglas de sintaxis y ortografía propias del lenguaje de programación utilizado; del BASIC en nuestro caso.

Las operaciones matemáticas básicas realizadas por un ordenador, instruido con el lenguaje BASIC, coinciden con las habituales en una calculadora: suma, resta, multiplicación, división y potenciación.

Los signos u *operadores* asociados a cada una de las citadas operaciones son los que muestran la tabla adjunta.

TABLA DE CONVERSION							
Ordenador	LIST					REM	
	LIST	LIST nl	LIST -nl	LIST nl-	LIST nl1 — nl2	REM	Signo equivalente
AMSTRAD	LIST	LIST nl	LIST -nl	LIST nl-	LIST nl1 - nl2	REM	'
APPLE II (APPLESOFT)	LIST	LIST nl	LIST ,nl	LIST nl,	LIST nl1,nl2	REM	—
APRICOT (M9BASIC)	LIST	LIST nl	LIST -nl	LIST nl-	LIST nl1 - nl2	REM	'
ATARI	LIST	LIST nl	LIST -nl	LIST nl,	LIST nl1,nl2	REM	—
CBM 64	LIST	LIST nl	LIST -nl	LIST nl-	LIST nl1 - nl2	REM	—
DRAGON	LIST	—	LIST -nl	LIST nl-	LIST nl1 - nl2	REM	'
EQUIPOS MSX	LIST	LIST nl	LIST -nl	LIST nl-	LIST nl1 - nl2	REM	—
HP-150	LIST	LIST nl	LIST -nl	LIST nl-	LIST nl1 - nl2	REM	Signo equivalente
IBM PC	LIST	LIST nl	LIST -nl	LIST nl-	LIST nl1 - nl2	REM	Signo equivalente
MPF	LIST	LIST nl	LIST -nl	LIST nl,	LIST nl1,nl2	REM	—
NCR DM-V (MS-BASIC)	LIST	LIST nl	LIST -nl	LIST nl-	LIST nl1 - nl2	REM	'
NEW BRAIN	LIST [-]	LIST nl	LIST -nl	LIST nl-	LIST nl1 - nl2	REM	—
ORIC	LIST	LIST nl	—	—	LIST nl1 - nl2	REM	Signo equivalente
SHARP MZ-700 (MZ-BASIC)	LIST	LIST nl	LIST -nl	LIST nl-	LIST nl1 - nl2	REM	—
SINCLAIR QL	LIST	LIST nl	LIST TO nl	LIST nl TO	LIST nl1 TO nl2	REMark	—
SPECTRAVIDEO	LIST	LIST nl	LIST -nl	LIST nl-	LIST nl - nl2	REM	—
ZX-SPECTRUM	LIST	—	—	LIST nl	—	REM	—

nl = Número de línea. nl1 = Número de línea inicial. nl2 = número de línea final.

FORMULACIONES DE LOS COMANDOS

LIST: Lista el programa completo. LIST nl: Muestra en pantalla la línea solicitada. LIST -nl: Presenta las líneas del programa comprendidas desde la primera hasta la especificada. LIST nl-: Lista desde la línea indicada hasta el final del programa. LIST nl1 - nl2: Muestra las líneas del programa comprendidas entre las dos especificadas, ambas incluidas. REM: Introduce un comentario en el programa. La presencia de esta instrucción es ignorada por el ordenador durante la ejecución. Signo equivalente: Signo que puede sustituir a la palabra comando REM.

En el lenguaje BASIC, los operadores aritméticos se utilizan para establecer relaciones matemáticas entre los datos, dentro del argumento de las instrucciones.

Este cometido es extensivo a los dos modos básicos de formulación de las instrucciones: directo (sin número de línea) e indirecto (con número de línea, formando parte de un programa). Naturalmente, utilizando instrucciones directas, que obtienen del ordenador una respuesta inmediata, puede simularse el funcionamiento propio de una calculadora. Por ejemplo:

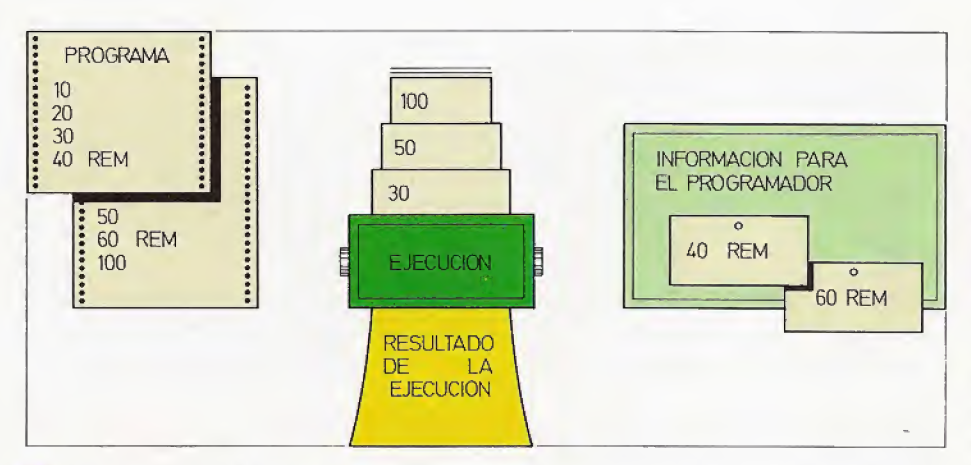
```
PRINT 2+4 (RT)
6
PRINT 5-6 (RT)
-1
PRINT 3*8 (RT)
24
█
```

Resulta obvio que para obtener la visualización de los resultados, hay que apoyarse en instrucciones de tipo PRINT. Su argumento es el que servirá para definir la operación a realizar.

Desde luego, es posible definir operaciones combinadas en las que intervengan varios datos y operadores.

El empleo de varios operadores aritméticos dentro de una misma expresión se rige por las siguientes normas:

- Dentro de cada expresión las operaciones se ejecutan siguiendo un orden perfectamente establecido: primero se opera la elevación a potencia, luego la multiplicación y/o división y, finalmente, la suma y/o resta.
- Si es necesario alterar la prioridad



El comando REM permite al programador introducir comentarios dentro de los programas. Más tarde serán una eficaz ayuda si es necesario variar alguna sentencia o estructura.

de las operaciones, por ejemplo, ejecutar una suma antes de una multiplicación, pueden utilizarse paréntesis. Las operaciones encerradas dentro de paréntesis tienen prioridad máxima.

- Si dentro de una misma expresión intervienen varias operaciones de igual prioridad, el intérprete BASIC las operará de izquierda a derecha.

Los siguientes ejemplos, ilustran la aplicación práctica de las prioridades que impone el BASIC:

```
5+3*2=11          (4+2)↑2=36
4+2/2=5           6+4/2*7=20
4+2↑2=8           6+4/(2*7)=6,29
(5+3)*2=16        9-5*8↑4/2=-10.231
(4+2)/2=3         9-5*8↑(4/2)=-311
```

A la hora de emular el funcionamiento de una calculadora ejecutando instrucciones en modo directo, puede recurrirse al empleo de variables. Por ejemplo, el siguiente par de instrucciones visualiza el resultado de una suma operada por medio de una asignación:

```
A=2+4 (RT)
PRINT A (RT)
6
█
```

En todo caso, es obvio que las posibilidades de cálculo del BASIC no están concebidas para realizar simples operaciones en modo directo. Su verdadero destino es la entrada en los argumentos de instrucciones indirectas que darán cuerpo a programas adecuados para resolver tareas más completas y evolucionadas.

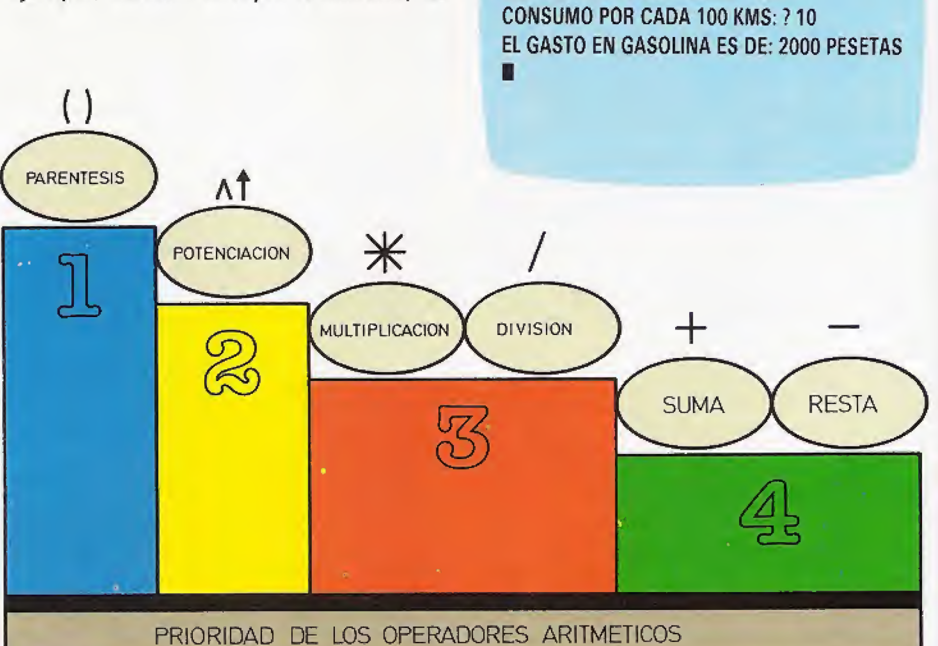
El programa que sigue constituye un ejemplo, sencillo aunque ilustrativo, de una aplicación de cálculo apoyada en la utilidad de los operadores aritméticos:

```
10 INPUT "PRECIO DEL LITRO DE GASOLINA:"; P
20 INPUT "KILOMETROS A RECORRER:";K
30 INPUT "CONSUMO POR CADA 100 KMS:";G
40 GA=P*K*G/100
50 PRINT "EL GASTO EN GASOLINA ES DE: ";GA;"PESE
   TAS"
60 END
```

El cometido del programa consiste en calcular cuál va a ser el gasto en gasolina necesario para recorrer un determinado trayecto en automóvil. Los datos que solicitará el ordenador para realizar el cálculo son: precio del litro de gasolina, número de kilómetros a recorrer y consumo de gasolina por cada 100 Kms. (por supuesto, del automóvil que vaya a utilizarse en el viaje). Estos tres datos, deben introducirse a medida que los solicite el ordenador, según vaya ejecutando la sucesivas instrucciones INPUT (líneas 10, 20 y 30).

```
RUN

PRECIO DEL LITRO DE GASOLINA: ? 80
KILOMETROS A RECORRER: ? 250
CONSUMO POR CADA 100 KMS: ? 10
EL GASTO EN GASOLINA ES DE: 2000 PESETAS
█
```



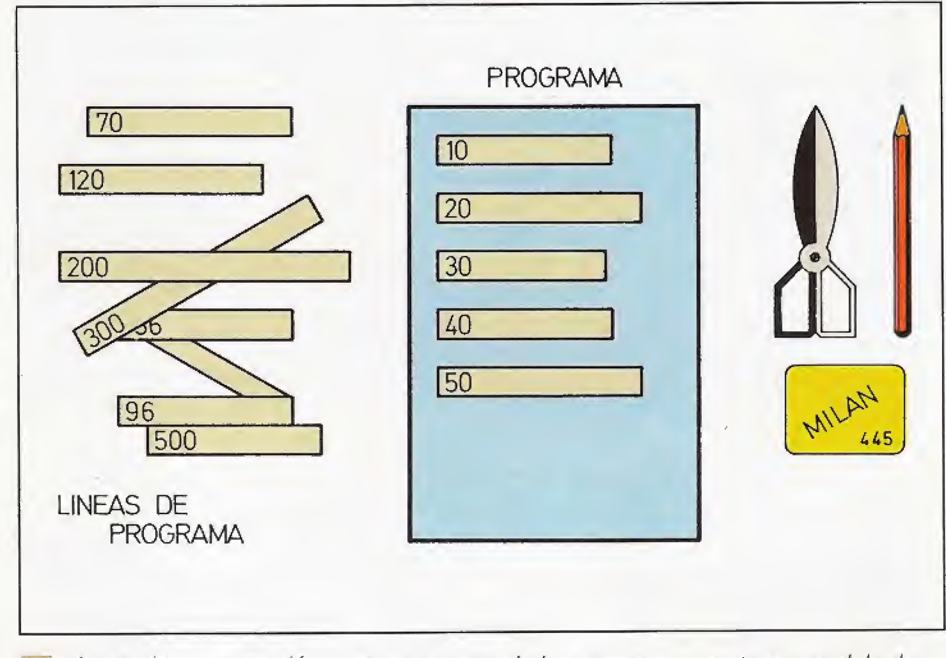
Al encontrarse con expresiones en las que se combinan datos por medio de operadores aritméticos, el ordenador ejecutará las operaciones de acuerdo a la prioridad establecida por el lenguaje BASIC. Las operaciones encerradas entre paréntesis son las que gozan de máxima prioridad.

Edición de programas

Escritura, corrección y puesta a punto de programas BASIC



A lo largo de los capítulos precedentes se han presentado algunos de los comandos que forman parte del vocabulario del lenguaje BASIC. Comandos cuyo objetivo es construir las instrucciones que darán cuerpo a los programas destinados a «educar» al ordenador. En este punto de la obra cabe plantearse un interrogante: ¿cómo hay que escribir los programas de forma que se aprovechen al máximo las posibilidades de edición que brinda el BASIC?



La escritura, corrección y puesta a punto de los programas son tareas englobadas en el apartado de «edición». Esta es una actividad apoyada por el propio traductor del lenguaje BASIC, a través de los denominados comandos de edición.

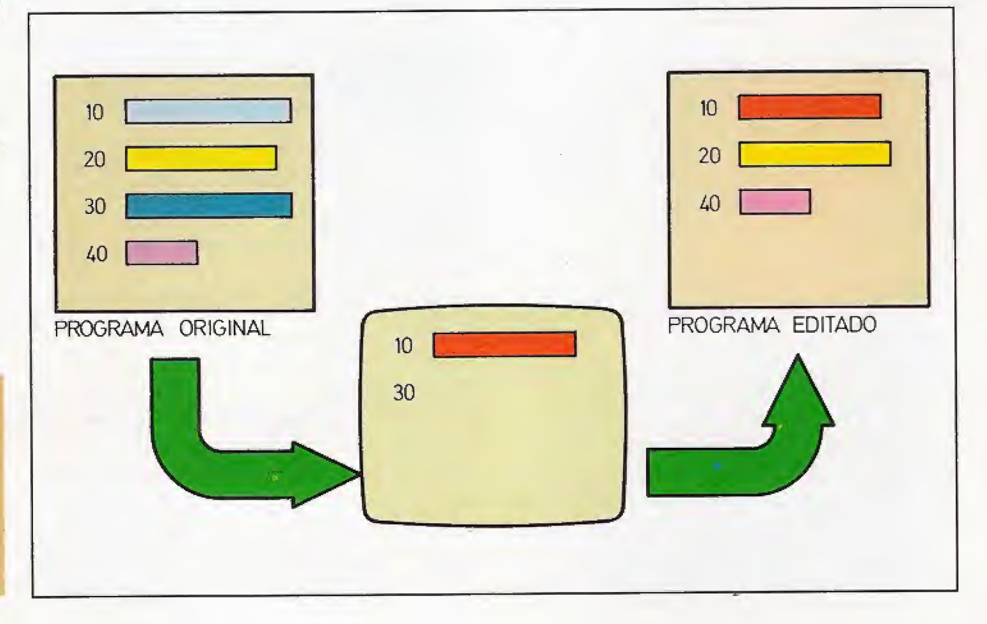
Escritura de un programa

La sesión de trabajo de un programador empieza, ineludiblemente, con la introducción del programa. Un programa que puede ser tan sencillo como el que aparece en la pantalla:

```
10 REM ENTRADA DE DATOS
20 INPUT "TATO 1"; A
30 PRINT "DATO 1"; A
40 PRINT "EL DATO ES:"; A
█
```

Cuatro líneas de programa cuya introducción se ha realizado ordenadamente y poniendo en práctica la misma secuencia en cada caso: teclear el correspondiente número de línea, su contenido y, por último, darla por concluida con una acción sobre la tecla de retorno (RETURN o ENTER).

El programa está ya en el interior de la máquina y puede ordenarse su ejecución con la orden RUN. Sin embargo, es conveniente comprobar antes su corrección. Tal como suele ocurrir con fre-



El procedimiento más elemental para la corrección de errores es el que refleja el gráfico adjunto. Para corregir una línea de programa, basta con reescribirla precedida por el mismo número. Si hay que borrarla, será suficiente con escribir su número de línea seguido de una acción sobre la tecla RETURN.

EDIT

Da paso al editor. Permite modificar el contenido de las líneas de programa.

Formato: EDIT [<*número de línea*>]

Ejemplos: EDIT
EDIT 30

EDITOR DE PANTALLA

10
20
30
40
50

DESPLAZAMIENTO
DEL CURSOR

Uno de los tres tipos básicos de editores es el de «pantalla». El usuario puede desplazarse a cualquier punto de la misma utilizando las teclas del cursor. Una vez colocado éste, puede introducir las oportunas modificaciones.

cuencia, se ha deslizado un error en su introducción; un error casi inapreciable (una letra alterada en la línea 20), pero suficiente para inutilizar la eficacia práctica del programa. ¿Cómo corregirlo? Aplicando el procedimiento de edición más elemental, será preciso escribir de nuevo la línea en cuestión, esta vez procurando que no se repita el error.

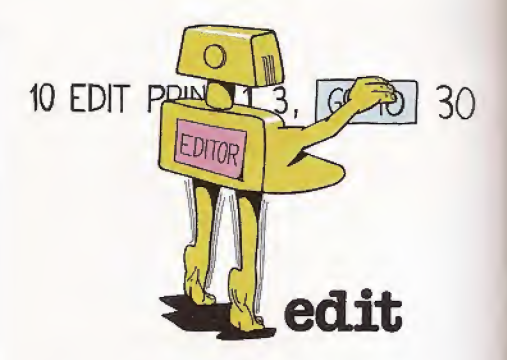
La línea escrita, cuyo número coincide con el de la que incluía el error, sustituirá a aquella en la memoria del ordenador. Esta circunstancia es fácilmente comprobable, sin más que ordenar la presentación en pantalla del listado por medio del comando LIST:

```
20 INPUT "DATO 1"; A

LIST
10 REM ENTRADA DE DATOS
20 INPUT "DATO 1"; A
30 PRINT "DATO 1"; A
40 PRINT "EL PRIMER DATO ES:"; A
■
```

En efecto, el error se ha subsanado: la palabra DATO de la línea 20 aparece ya corregida.

Un nuevo repaso al programa puede decidir al programador por la supresión de la línea 30, cuya presencia es redundante. De nuevo, puede aplicarse el mismo procedimiento: escribir una nueva línea de programa precedida por el mismo número. En este caso, como quiera que se trata de eliminar su presencia, basta con escribir sólo el número de la línea afectada y pulsar la tecla RETURN a continuación. El resultado



queda patente al utilizar una vez más el comando LIST:

```
30 (CR)

LIST
10 ENTRADA DE DATOS
20 INPUT "DATO 1"; A
40 PRINT "EL PRIMER DATO ES:"; A
■
```

Editores de programas

Una sesión de trabajo como la relatada en el apartado precedente puede convertirse en una tarea prolongada y tediosa. Normalmente, los programas a editar serán mucho más extensos y sus instrucciones bastante más complejas. En tal caso, es obvio que la repetición del contenido total de las líneas a modificar no será una actividad grata ni eficaz.

Afortunadamente, la mayor parte los traductores de lenguaje BASIC disponen de herramientas auxiliares para corregir errores y realizar modificaciones con mayor comodidad y rapidez. El conjunto de medios que brinda el ordenador para la edición de programas recibe el nombre de *editor*.

Existen muchos tipos de editores, diferenciados por su potencia y por las posibilidades que ponen a disposición del programador. En todo caso, el aprendizaje del manejo del editor es de suma importancia a la hora de adquirir un or-

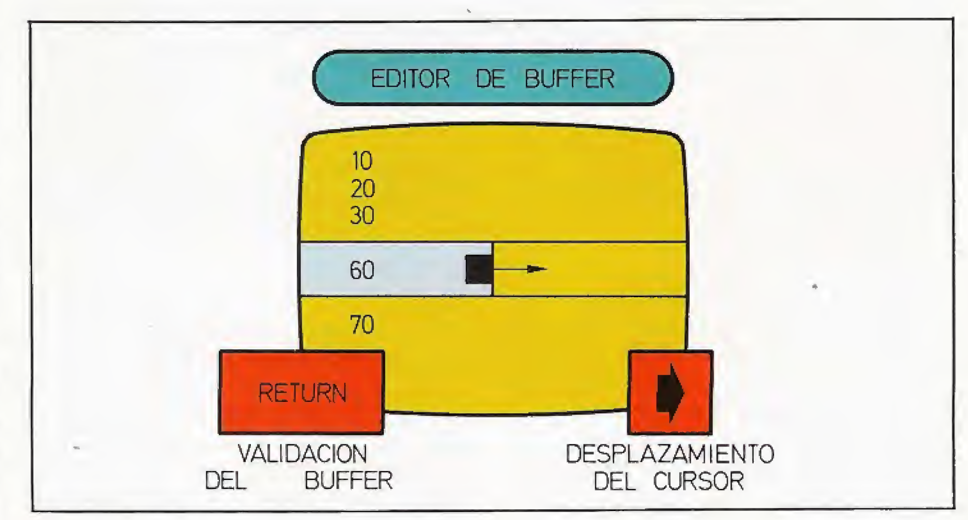
denador y disponerse a confeccionar programas para el mismo.

En muchos ordenadores está permitido realizar modificaciones y corregir errores (editar) inmediatamente después de haber listado la porción afectada del programa. Normalmente, se puede acceder a la línea deseada accionando las teclas para el desplazamiento del cursor. Una vez posicionado el cursor se procede a modificar la línea, ya sea insertando caracteres o bien realizando una nueva escritura encima de la zona a corregir. Después de realizar los cambios pertinentes, basta con pulsar la tecla RETURN en cualquier punto de la línea para que se haga definitiva la modificación realizada.

Este tipo de editor se denomina «full screen» o de *pantalla completa,* debido a que la edición puede realizarse en cualquier punto de la pantalla.

Los editores de pantalla presentan algunas limitaciones en ciertos casos. Por ejemplo, hay equipos que permiten introducir caracteres de control dentro del texto asociado a una instrucción PRINT; caracteres que se obtienen directamente mediante el uso de las teclas del cursor y otras especiales. De esta forma, cuando se está escribiendo una constante alfanumérica (ello viene señalado por las comillas que preceden al texto escrito), las teclas del cursor no realizarán la acción prevista, sino que escribirán el carácter de control correspondiente. Ello impedirá el libre movimiento del cursor en dicha zona. Otros ordenadores incorporan un segundo tipo de editor: el denominado *editor de líneas,* cuya actividad está regida por el comando EDIT. En este caso, sólo es posible editar la línea seleccionada por medio del comando EDIT. La selección se realiza colocando el número de la línea a editar tras el referido comando. A continuación, pueden ya realizarse las correcciones oportunas, apoyándose en el repertorio de subcomandos del editor.

El «editor de líneas» reduce el campo de trabajo a la línea seleccionada mediante el comando EDIT. La modificación de su contenido se realiza desplazando el cursor a lo largo de la misma y utilizando los subcomandos de edición oportunos.



El editor menos evolucionado y más incómodo para el usuario es el de «buffer». Los caracteres de la línea en edición barridos por el cursor y las modificaciones introducidas, se copian en una memoria temporal. El contenido de esta última será aceptado como definitivo al accionar la tecla RETURN.

El comando EDIT

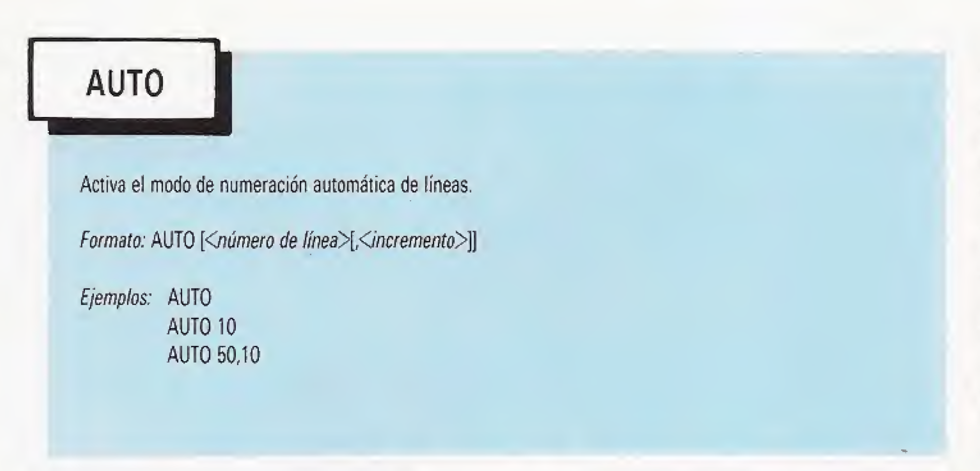
EDIT es el comando básico en torno al que se organiza el funcionamiento del *editor de líneas.* Al ejecutarlo, se accede al denominado *modo de edición:* situación que permite operar las modificaciones necesarias en la línea de programa seleccionada.

El formato de una instrucción del tipo EDIT es el siguiente:

EDIT <número de línea>

El número de línea debe coincidir con el de la línea de programa a editar.

Tras la ejecución de este comando, la línea seleccionada entra en modo edición. Algunos editores de líneas muestran la línea indicada en la zona inferior de la pantalla, lista para su edición. En otros, únicamente aparecerá el número

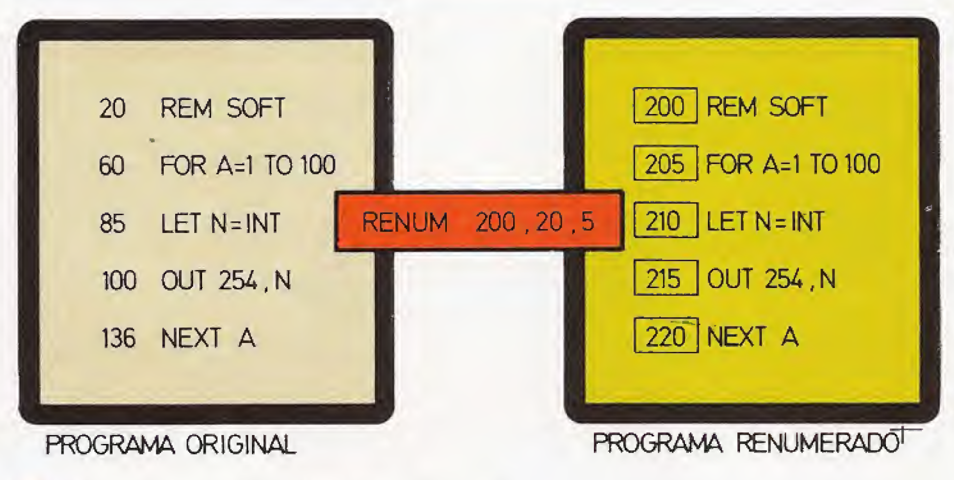


AUTO

Activa el modo de numeración automática de líneas.

Formato: AUTO [<*número de línea*>[,<*incremento*>]]

Ejemplos: AUTO
AUTO 10
AUTO 50,10



Renumeración automática de un programa por efecto de una instrucción RENUM.

de la línea de edición y el cursor situado inmediatamente después. Acto seguido, será necesario seleccionar una de las diversas opciones de edición disponibles.

El editor de líneas exige teclear el comando EDIT cada vez que se desee editar una nueva línea. No obstante, una vez que se ha entrado en modo edición se permite el libre movimiento del cursor a través de la línea seleccionada. En esta situación, algunas teclas específicas dan paso a las funciones de inserción, borrado o sustitución de caracteres.

El editor de «buffer»

Existe un tercer tipo de editor, tal vez el más primitivo de los mencionados.

Para manejarlo es preciso situarse al principio de la línea a editar (en cualquier punto de la pantalla), e ir «copiando» los caracteres en un «buffer» o memoria temporal. Esto se consigue «pasando por encima» de los mencionados caracteres con el cursor. A medida que se van dando las órdenes adecuadas, se consigue la inserción o el borrado de los caracteres necesarios.

El principal inconveniente de los editores que utilizan buffer es la necesidad de repasar toda la línea. Esta característica obliga a llegar al final de la misma antes de poder accionar la tecla RETURN.

Semejante imperativo deriva de la forma en la que se aceptan los caracteres en edición. El editor dispone al efecto de una zona de memoria denominada «buffer», de la cual se extrae la línea editada al concluir el proceso de edición accionando la tecla RETURN.

Para introducir caracteres en el buffer es necesario pasar el cursor sobre ellos. Si se teclea algún carácter nuevo, éste entrará en el buffer inmediatamente. La inserción o sustitución de carateres se habilita por medio de las teclas de desplazamiento del cursor, posicionando a éste en el lugar adecuado. Ello permite saltar porciones de la línea que no serán introducidas en el buffer. No cabe duda que este método de edición exige una profunda concentración por parte del programador; éste ha de llevar en mente los caracteres introducidos en el buffer.

La diversidad de los editores

En general, el editor de uso más fácil es el de pantalla, puesto que permite una edición inmediata. Permite corregir cualquier instrucción presente en pantalla sin más que accionar la tecla de retroceso de carro tras realizar las correcciones oportunas.

Los editores que utilizan un buffer son los más incómodos y cuyo empleo es menos inmediato; no obstante, una vez que su manejo resulta familiar constituirán también una eficaz herramienta de trabajo. Los editores de línea son, en principio, los menos potentes; sin embargo, de ellos se puede conseguir mucho más de lo que parece a primera vista. Existen muchas variantes, aunque, por lo general, hay que entrar en el modo de edición por medio de una orden específica (normalmente EDIT).

Al margen del tipo de editor que incorpore cada dialecto BASIC, existen ciertos comandos de apoyo a las tareas de edición de programas que pueden estar presentes en distintos equipos. Este grupo de comandos resultan adecuados para numerar automáticamente las sucesivas líneas de un programa en edición, para su renumeración automática, o para el borrado directo de líneas de programa.

El comando AUTO

A menudo resulta tedioso el hecho de tener que teclear el número de línea cuando se introduce un programa por

vez primera. Incomodidad que aumenta a medida que el programa es más extenso. Para facilitar esta tarea, algunos traductores BASIC disponen del comando AUTO. El referido comando libera al usuario de la necesidad de llevar la cuenta de los números de línea, puesto que se encarga de numerar las líneas a medida que se introducen, calculando el número correspondiente a cada línea sucesiva.

El formato de una instrucción construida a partir del comando AUTO es el siguiente:

AUTO [<número de línea>[,<incremento>]]

Una vez ejecutado en modo directo, el ordenador generará de forma automática el número de línea correspondiente tras cada acción del programador sobre la tecla de retroceso de carro (CR).

AUTO comienza la numeración del progama a partir del número indicado en el argumento: <número de línea>. Este valor crecerá en las sucesivas líneas en el número de unidades que se hayan indicado en la zona de <incremento>.

Cuando la instrucción AUTO se formula desprovista de argumento, el ordenador considera que ambos valores (<número de línea> e <incremento>) adoptan el valor 10. Por ello, numerará el programa a partir de la línea 10 y en sucesivos incrementos de 10 unidades.

Si el <número de línea> va seguido por una coma y no se especifica el incremento, se entiende que el valor del incremento es igual al incremento indicado en el último comando AUTO ejecutado con anterioridad.

El comando RENUM

A la hora de intentar pulir y perfeccionar un determinado programa BASIC, es fundamental entender la función de cada zona del mismo, así como las relaciones entre ellas. Para ello no basta con conocer a fondo el lenguaje. Muchas veces, la complejidad de su estructura no pemite un seguimiento fácil de la ejecución.

La inserción de comentarios (REM) suele aclarar el cometido de los distintos fragmentos del programa. Otro método clarificador consiste en agrupar las líneas del programa en bloques fraccionados. Por ejemplo, una cierta rutina o zona de cálculo específica se puede situar en las líneas 2000 y siguientes, otra a partir de la 3000, etc. Posteriormente, será posible identificar las distintas partes del programa atendiendo exclusivamente a las primeras cifras de los números de línea. Habitualmente, las líneas intercaladas contribuyen al desorden general, alterando la pauta de numeración. En determinados casos, incluso se hace imposible insertar una nueva línea: cuando los números de dos líneas consecutivas difieren en una sola unidad.

Ambas situaciones hacen aconsejable —cuando no obligan— «renumerar» las líneas del programa; esto es: cambiar los números de las líneas sin alterar el orden de las mismas.

El comando utilizable a tal efecto es RENUM. Este permite sustituir los números de línea contiguos por otros nuevos, especificando, además, el salto deseado entre cada dos números consecutivos. Su formato general es:

RENUM [[<nuevo>][,[<anterior>]-[,<incremento>]]]

Como se ha indicado, su misión es la de sustituir los números de línea situados a partir del indicado de la zona <anterior>, hasta el final del programa, por los números <nuevo> y sucesivos, en saltos de tantas unidades como especifique la zona de <incremento>.

El número indicado en el campo <anterior> hace referencia al primer número de línea a sustituir. Semejante precisión permite renumerar sólo una parte del programa, manteniendo en otras zonas la numeración de líneas original.

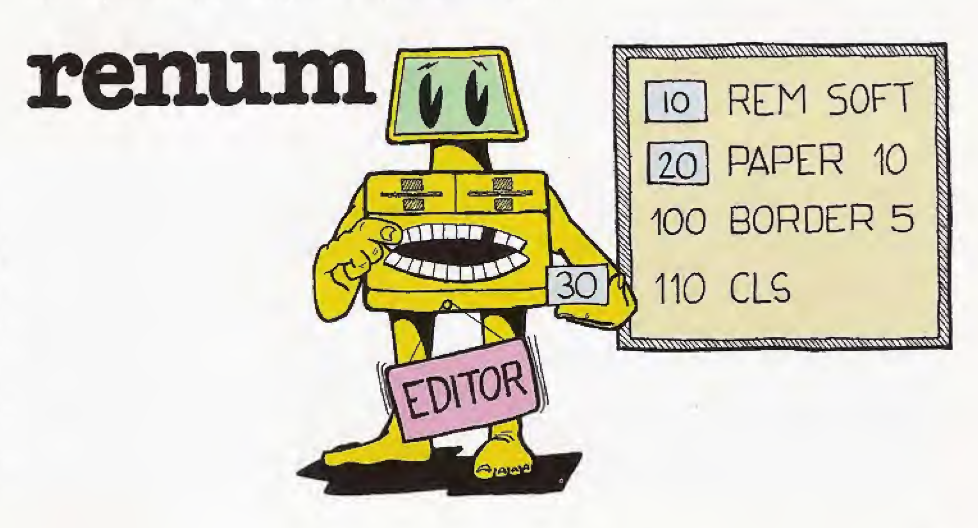
El siguiente ejemplo ilustra la actuación de este tipo de instrucciones. El objeto de trabajo es un simple programa capaz de calcular el cuadrado del número que se introduzca como respuesta a la instrucción INPUT. El programa finalizará al recibir un cero como dato de entrada.

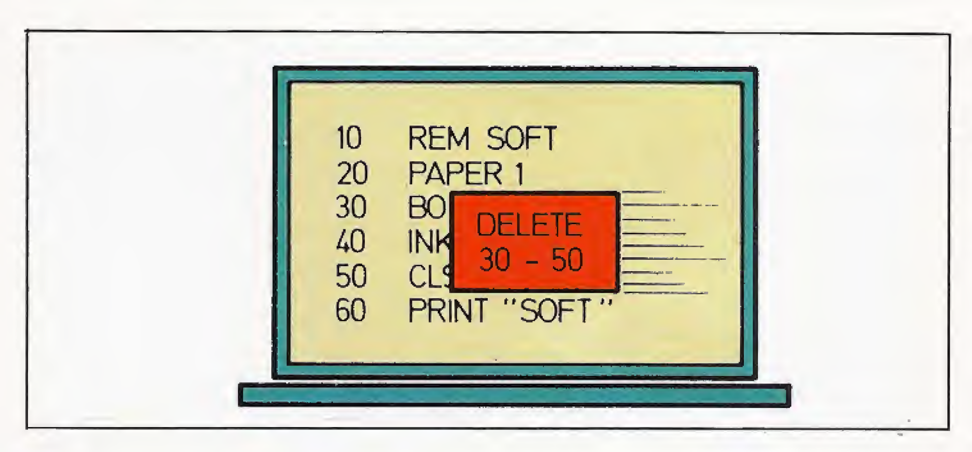
RENUM

Renumera líneas de programa.

Formato: RENUM [[*<número nuevo>*] [,[*<número antiguo>*] [,*<incremento>*]]]

Ejemplos: RENUM 100
RENUM 100,10
RENUM 100,10,5





El comando DELETE permite borrar una línea o un grupo de ellas «de un plumazo». El bloque de líneas que se desea eliminar se define en el argumento de la referida instrucción.

```
 10 REM COMANDO RENUM
 20 PRINT "PROGRAMA EJEMPLO"
 21 INPUT A
 22 PRINT
 23 LET B=A*A
122 PRINT "EL CUADRADO DE";A
157 PRINT "ES:";B
158 END
█
```

Un programa de indudable sencillez, pero que ganaría en claridad con la inserción de algunos comentarios.

El problema surge en la peculiar numeración de sus líneas. ¿Cómo es posible emplazar una instrucción entre las líneas 20 y 21, o entre la 21 y 22? No queda más alternativa que abrir espacio entre las mencionadas líneas; una solución que el comando RENUM pondrá en práctica de forma inmediata y automática:

```
RENUM 100,21,2

LIST
 10 REM COMANDO RENUM
 20 PRINT "PROGRAMA EJEMPLO"
100 INPUT A
102 PRINT
104 LET B=A*A
106 PRINT "EL CUADRADO DE";A
108 PRINT "ES:";B
110 END
```

La instrucción RENUM formulada, ordena la renumeración de la zona de programa situada a partir de la línea 21. La nueva numeración debe partir del número de lína 100 y distribuirse en sucesivas líneas cuyos números se distancien en incrementos de 2 unidades. La ejecución de una orden LIST permite observar la nueva distribución de números de línea.

Ahora es posible ya emplazar nuevas instrucciones precediendo a las instrucciones INPUT y PRINT. Por ejemplo:

```
 95 PRINT "INTRODUZCA UN NUMERO"
101 REM

LIST
 10 REM COMANDO RENUM
 20 PRINT "PROGRAMA EJEMPLO"
 95 PRINT "INTRODUZCA UN NUMERO"
100 INPUT A
101 REM
102 PRINT
104 LET B=A*A
106 PRINT "EL CUADRADO DE";A
108 PRINT "ES:";B
110 END
```

Un nuevo uso de RENUM permitirá remodelar la numeración de líneas logrando una districución consecutiva en saltos de diez unidades:

```
RENUM 30,95,10

LIST
 10 REM COMANDO RENUM
 20 PRINT "PROGRAMA EJEMPLO"
 30 PRINT "INTRODUZCA UN NUMERO"
 40 INPUT A
 50 REM
 60 PRINT
 70 LET B=A*A
 80 PRINT "EL CUADRADO DE";A
 90 PRINT "ES:";B
100 END
```

El programa está ya completo y organizado con la colaboración del comando RENUM. Un comando cuya actuación va más allá de la simple modificación de los números que preceden a las líneas de programa: modifica también automáticamente todas las instrucciones de saltos y llamadas a subrutinas.

Borrado de líneas

A la hora de depurar un programa, tan importante es la posibilidad de intercalar nuevas líneas como la de borrar las antiguas. Esta segunda alternativa se consigue por el simple procedimiento de

DELETE

Elimina las líneas situadas entre las especificadas.

Formato: DELETE [<*línea inicial*>]–[<*línea final*>]

Ejemplos: DELETE 100–225
DELETE 100–
DELETE –225

TABLA DE CONVERSION

Ordenador	EDIT	AUTO	RENUM	DELETE
	EDIT <nl>	**AUTO <nl>, <inc>**	**RENUM <nue.>, <ant.>, <inc.>**	**DELETE <inic.>-<fin.>**
AMSTRAD	EDIT <nl>	AUTO <nl>, <inc>	RENUM <nue>, <ant.>, <inc.>	DELETE <inic.>-<fin>
APPLE II (APPLESOFT)	—	—	—	DELETE <inic.>-<fin.>
APRICOT (M-BASIC)	EDIT <nl>	AUTO <nl>, <inc.>	RENUM <nue.>, <ant.>, <inc.>	—
ATARI	—	—	—	—
CBM 64	—	—	—	DEL <inic.>-<fin.>
DRAGON	EDIT <nl>	—	RENUM <nue.>,<ant.>,<inc.>	DELETE <inic.>-<fin.>
EQUIPOS MSX	—	AUTO <nl>, <inc.>	RENUM <nue.>, <ant.>, <inc.>	DELETE <inic.>-<fin.>
HP-150	EDIT <nl>	AUTO <nl>, <inc.>	RENUM <nue.>, <ant.>, <inc.>	DELETE <inic.>-<fin.>
IBM PC	EDIT <nl>	AUTO <nl>, <inc.>	RENUM <nue.>, <ant.>, <inc.>	DELETE <inic.>-<fin.>
MPF	—	—	—	DEL <inic.>-<fin.>
NCR DM-V (MS-BASIC)	EDIT <nl>	AUTO <nl>,<inc.>	RENUM <nue.>, <ant.>, <inic.>	DELETE <inic.>-<fin.>
NEW BRAIN	—	—	—	DELETE <inic.>-<fin.>
ORIC	EDIT <nl>	AUTO <nl>, <inc.>	—	—
SHARP MZ-700 (MZ-BASIC)	—	AUTO <nl>, <inc.>	RENUM <nue.>, <ant.>, <inc.>	DELETE <inic.>-<fin.>
SINCLAIR QL	EDIT <nl>, <inc.>(1)	AUTO <nl>, <inc.>	RENUM <inic.>TO<fin.>; <nue.>, <inc.>(1)	DLINE <inic.>TO<fin.>(1)
SPECTRAVIDEO	—	AUTO <nl>, <inc.>	RENUM <nue.>, <ant.>, <inc.>	DELETE <inic.>-<fin.>
ZX-SPECTRUM	EDIT (2)	—	—	—

<nl>: Número de línea. <inc.>: Incremento. <nue.>: Nuevo número de línea. <ant.>: Número de línea antiguo. <inic.>: Número de línea inicial. <fin.>: Número de línea final.

FORMULACIONES DE LOS COMANDOS

EDIT <NL>: Edita la línea cuyo número se indica. AUTO <nl>, <inc.>: Genera automáticamente los sucesivos números de línea tras cada pulsación de la tecla RETURN. La numeración empieza a partir de <nl>, en sucesivos incrementos de <inc.> unidades. RENUM <nue.>, <ant.>, <inc.>: Renumera las líneas del programa a partir de la indicada en <ant.>, situándolas a partir de la línea <nue.> en incrementos de <inc.> unidades. DELETE <inic.>-<fin.>: Borra el bloque de líneas comprendidas entre las dos especificadas.

OBSERVACIONES

(1) SINCLAIR QL

EDIT <nl> [, <inc.>]: Edita la línea <nl> y sucesivas en incrementos de <inc.>. RENUM [<INIC.> [TO <fin>;]][<nue>][, <inc.>: Renumera automáticamente desde la línea <inic.> hasta la <fin.>, tomando como base de la nueva numeración la línea <nue.> en incrementos de <inc.> unidades. DLINE <inic.> TO <fin.>: Borra el bloque de líneas indicado. Pueden encadenarse expresiones de este tipo, separando por comas los rangos de líneas a borrar.

(2) ZX-SPECTRUM

El comando EDIT no lleva asociado un número de línea. No obstante, hay que situar un cursor que aparece en pantalla sobre la línea a editar.

Subcomandos del editor de líneas

Los editores de líneas suelen disponer de un conjunto de subcomandos que facilitan la edición de la línea en curso. Estos subcomandos se detallan a continuación, agrupados de acuerdo a la función que realizan:

CURSOR

Espacio. Mueve el cursor un lugar hacia la derecha, mostrando el carácter que ocupa dicha posición en la línea original.

INSERCION

I. Permite insertar tantos caracteres como se desee a partir de la posición en la que esté situado el cursor. Para salir de esta opción es necesario pulsar la tecla ESCAPE.
X. Extensión de línea. Desplaza el cursor al final de la línea y entra en inserción en ese punto.

BORRADO

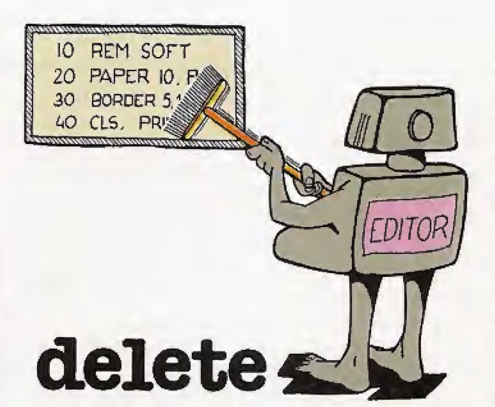
D. Borra el carácter situado en la posición siguiente a la que ocupa el cursor.
H: Borra todos los caracteres situados a la derecha del cursor y entra automáticamente en modo inserción.
S: Va acompañado de un número y un carácter. Buscará el carácter indicado tantas veces como señale el número introducido y colocará el cursor delante del carácter afectado.
K: Actúa de forma análoga al subcomando S, si bien, borra todos los caracteres por los que pasa.

SUSTITUCION

C: Sustituye el carácter situado a la derecha del cursor por el que se introduzca a continuación de C; puede acompañarse por un parámetro que indicará el número de caracteres contiguos a sustituir.

FINAL

(CR): Señala el final del proceso de edición; al accionar la tecla en cuestión (RETURN o ENTER) el ordenador aceptará todos los cambios realizados.
E: Salida de las modificaciones introducidas.
L: Lista la línea en curso y entra en edición la siguiente.



teclear el número correspondiente a la línea en cuestión, seguido por una acción sobre la tecla RETURN. Ello equivale a sustituir la línea especificada por una línea vacía de contenido.

Este método resulta útil y rápido cuando la cantidad de líneas a borrar no es muy grande. Sin embargo, cuando se trata de borrar amplias zonas del programa, el trabajo puede llegar a resultar agotador. Para este cometido existe un comando BASIC experto en el borrado de varias líneas a la vez: DELETE. Su formato como instrucción directa es el siguiente:

```
DELETE <inicial>—<final>
```

Los campos que siguen al comando determinan las líneas que deben desaparecer del programa. El primer número especifica el número de línea *inicial* o inferior del bloque a eliminar, y el segundo el número de línea *final* o mayor de los afectados por el borrado.

Partiendo del último listado remodelado por medio del comando RENUM, puede observarse el efecto de la omisión de uno de los campos en la instrucción DELETE.

```
DELETE 30

LIST
 10 REM COMANDO RENUM
 20 PRINT "PROGRAMA EJEMPLO"
 40 INPUT A
 50 REM
 60 PRINT
 70 LET B=A*A
 80 PRINT "EL CUADRADO DE";A
 90 PRINT "ES:";B
100 END
```

Una de las posibles formulaciones parciales de la instrucción DELETE es la que incorpora un sólo número de línea en el argumento. Como revela el ejemplo anterior, su efecto se traduce exclusivamente en el borrado de la línea indicada.

Otra de las posibles variantes de DELETE es la que muestra el siguiente ejemplo (DELETE 80—). La ausencia de número de línea final supone el borrado de todas las líneas del programa localizadas a partir de la inicial (línea 80 en el ejemplo):

```
DELETE 80—

LIST
10 REM COMANDO RENUM
20 PRINT "PROGRAMA EJEMPLO"
40 INPUT A
50 REM
60 PRINT
70 LET B=A*A
```

Una nueva ejecución de la instrucción DELETE, esta vez desprovista de número de línea inicial (DELETE −20), conduce al borrado de las líneas que van desde el principio del programa hasta la indicada (línea 20):

```
DELETE—20

LIST
40 INPUT A
50 REM
60 PRINT
70 LET B=A*A
```

Por último, ya sólo queda por observar la actuación de DELETE con su formato de instrucción más genérico:

```
DELETE 50—60

LIST
40 INPUT A
70 LET B=A*A
█
```

Almacenamiento de programas

Grabación y lectura de programas en la memoria auxiliar

Un hecho fácilmente constatable es que una vez que se desconecta la alimentación del ordenador, la memoria interna de éste se borra y, en consecuencia, el programa que reside en su interior desaparece. Si se desea conservar un programa o un conjunto de datos, será necesario disponer de una unidad de almacenamiento masivo: una casete de audio o, mejor, un disco magnético.

MEMORIA CENTRAL
PROGRAMA
PROGRAMA
RAM

INTRODUCCION DEL PROGRAMA EN LA MEMORIA INTERNA DEL ORDENADOR

GRABACION DEL PROGRAMA EN UNA UNIDAD DE ALMACENAMIENTO EXTERNO

El objetivo habitual de un programa no se reduce a ejecutarlo una sola vez. En consecuencia, tras introducirlo en el ordenador a través del teclado es conveniente grabarlo en un soporte de memoria permanente (por ejemplo, en cinta magnética o disco).

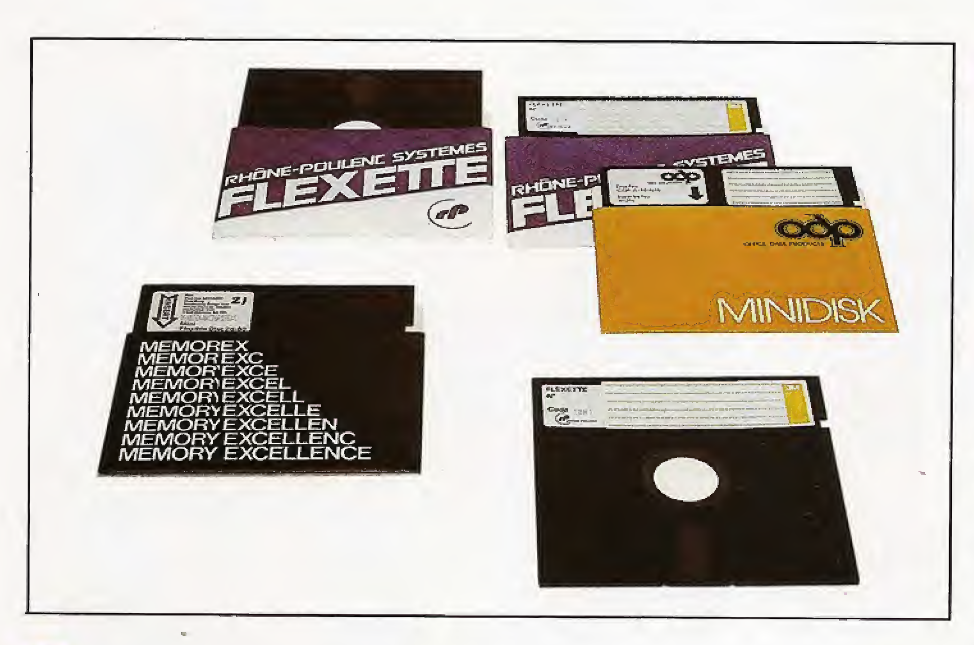
Grabación de programas en casete

Las unidades de casete constituyen una de las memorias de masa más utilizadas en equipos domésticos. El motivo fundamental radica en su economía. La simplicidad de estas unidades hace que resulten fáciles de construir y muy robustas. En muchos casos, incluso, no será necesario adquirir una de estas unidades, ya que su función puede realizarla un simple magnetófono para casete de audio de tipo convencional. Ello aporta algunas ventajas. Raro es el aficionado que no dispone de un magnetófono a casete, de ahí que el uso de estos periféricos para el almacenamiento masivo resulte muy asequible y, desde luego, económico.

También los soportes empleados, las cintas en casete, son baratas y fáciles de adquirir. En principio, cualquier cinta comercial de audio resulta adecuada, si bien, es conveniente huir de las cintas de escasa calidad, ya que pueden ocasionar problemas que conducirán a la pérdida de la información en ellas almacenada. Para empezar el trabajo con la unidad de casete, hay que contar con un programa, por ejemplo el siguiente:



Las unidades de casete constituyen el periférico de almacenamiento externo más económico. Su presencia es frecuente junto a los ordenadores personales de tipo doméstico. Algunos equipos deben utilizar una unidad de casete específicamente diseñada por el fabricante; otros ordenadores pueden trabajar asociados a cualquier magnetófono de audio de tipo común.



Las unidades de almacenamiento en cinta magnética se caracterizan por su acceso secuencial y su baja velocidad de transferencia. Si se desea elevar ésta, o conseguir un acceso directo a cualquier programa, es necesario utilizar una unidad de disco.



Una de las características propias de los lenguajes de programación es la de brindar al programador herramientas para facilitar su trabajo. LIST y REN son dos de los comandos que ofrece el BASIC para este cometido.

```
10 REM PROGRAMA DE PRUEBA
20 PRINT "PROGRAMA"
30 PRINT "DE DEMOSTRACION"
40 PRINT "PARA EL ALMACENAMIENTO"
50 PRINT "DE INFORMACION"
60 PRINT "EN CASETE"
■
```

Acto seguido, será preciso conectar la referida unidad al ordenador; para ello, es necesario consultar el manual de cada equipo, ya que en este punto no caben métodos generales.

Ahora puede ya realizarse la grabación del programa en casete. Para ello, es necesario que el ordenador transfiera una señal a la unidad de casete, señal que será grabada para su posterior uso. Desde luego, la transferencia no se va a realizar de una forma automática, sólo con que el usuario lo desee, es necesario comunicar la orden adecuada al equipo. Esta coincide, en muchos casos, con el comando BASIC CSAVE, cuya formulación es la siguiente:

```
CSVE"<nombre del fichero>"
```

El nombre del fichero es una referencia que identificará al fichero o bloque de datos (el programa en este caso) y que facilitará su manipulación. No hay que perder de vista que un mismo casete suele constituir el soporte de almacenamiento de más de un programa; por lo tanto, el nombre del fichero permitirá identificar a cada uno de los programas por separado. Dependiendo del ordenador del que se trate, el nombre debe cumplir determinadas condiciones; por lo general, éstas se refieren a la longitud máxima del mismo, habitualmente de seis a ocho caracteres. Por ejemplo:

```
CSAVE "CUENTA"
CSAVE "PROG"
```

En ciertos casos, es posible añadir un apellido al nombre del archivo. Este apellido es una extensión que acompaña al nombre y consiste en dos o tres ca-

racteres, situados a la derecha del nombre, y separados de éste por un punto.

Al ejecutar una instrucción CSAVE, el ordenador transferirá el programa que se encuentra en la memoria al casete, en forma de señal eléctrica. Esta contiene la información necesaria para que, posteriormente, sea posible realizar el proceso inverso, esto es: leer dicha señal y reconstruir el programa en el ordenador. Para ello será necesario que la unidad de casete se encuentre en condiciones de recibir la señal y grabarla. Al respecto, habrá que pulsar simultáneamente las teclas de grabación y avance del magnetófono. Desde luego, es preciso sincronizar esta operación con la introducción del comando.

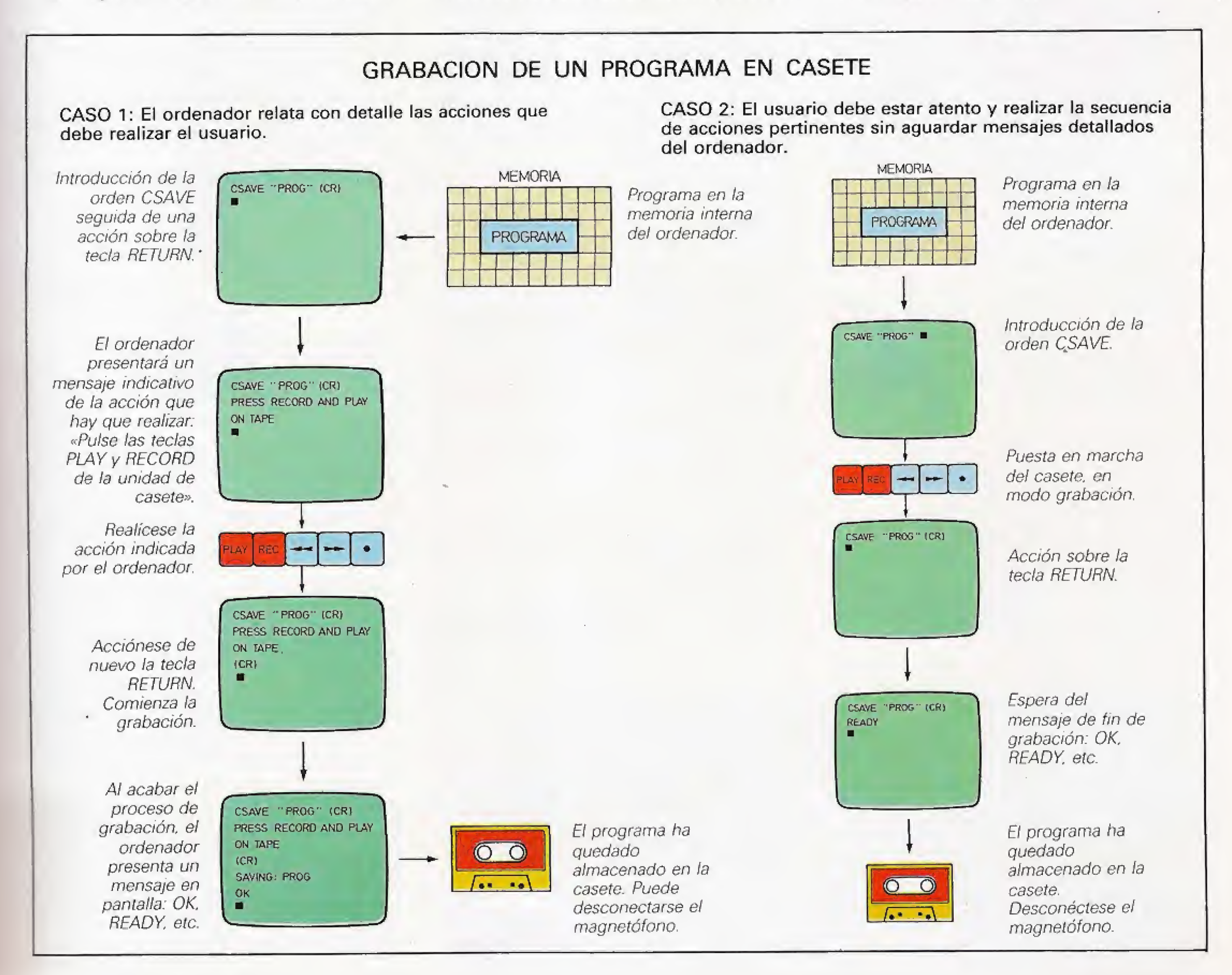
En algunas ocasiones, esta tarea es bastante sencilla; una vez introducido el comando, el ordenador pedirá al usuario que ponga a la unidad de casete en situación y, tras ello, que accione una tecla (RETURN, normalmente) para que dé comienzo el proceso de grabación.

El proceso de grabación no tiene una duración fija, sino que depende de la longitud del programa y también de la velocidad con la que el ordenador es capaz de enviar los datos a la unidad. El proceso puede durar desde unos segundos hasta varios minutos. En cualquier caso, no hay por qué preocuparse; el ordenador se encargará de informar al operario en el instante en el que termine la grabación, presentando en la pantalla el oportuno mensaje. Tras ello, es conveniente detener el casete para que no siga grabando; en algunos casos el propio ordenador se encargará de detenerlo.

Algunos ordenadores no permiten la grabación de un programa con nombre; de ocurrir así, será suficiente con introducir el comando de grabación sin argumento alguno: CSAVE.

Carga de programas desde casete

Cualquier programa grabado en cinta magnética, ya sea por el usuario mediante el comando CSAVE, o bien grabado por el fabricante del casete si se trata de un programa comercial, puede cargarse en el ordenador y ejecutarse en cualquier momento.



CSAVE

Almacena un programa en la unidad casete, otorgando al mismo el nombre especificado.

Formato: CSAVE "<*nombre del programa*>"

Ejemplo: CSAVE "CASA"



Las casetes y, en general, las cintas magnéticas son soportes de memoria de tipo secuencial. En ellas la información se graba en serie, dato a dato. De ahí que para acceder a un dato específico haya que pasar por todos los almacenados en posiciones precedentes.

La necesidad de almacenamiento de los programas es una realidad apuntada en párrafos anteriores. En cualquier momento puede desconectarse accidentalmente la alimentación del aparato, o el propio usuario, por error, puede introducir el comando NEW. ¿Qué sucede en tal caso? La respuesta es que la memoria del ordenador quedará «limpia». La ejecución de un comando LIST no obtendrá respuesta; lo mismo sucederá con el comando RUN. El programa se ha perdido. No obstante, si se tomó la precaución de grabarlo previamente, será posible recuperarlo en cualquier instante. De ahí que cuando se esté desarrollando un nuevo programa, resulte conveniente tomarse la molestia de hacer copias de seguridad en las sucesivas versiones, a medida que se vaya perfeccionando el mismo. De no hacerlo así, el usuario puede llevarse alguna que otra sorpresa, especialmente en lo que se refiere a la alimentación del aparato, siempre a expensas de la red.

El comando adecuado para ordenar la carga de un programa almacenado en cinta es, normalmente, CLOAD. Su aspecto general es el siguiente:

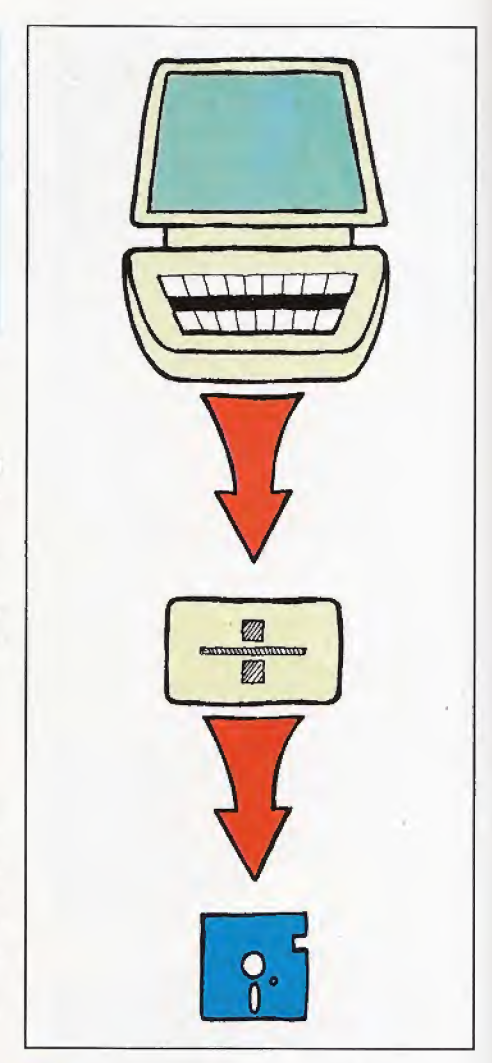
CLOAD"<nombre del programa>"

El nombre del programa ha de coincidir con el utilizado en el proceso de grabación. Sin embargo, si no se recuerda el nombre, éste puede omitirse; en tal caso, el ordenador cargará el primer programa que localice en la casete. A continuación, se muestran algunos ejemplos de formulaciones válidas del comando CLOAD

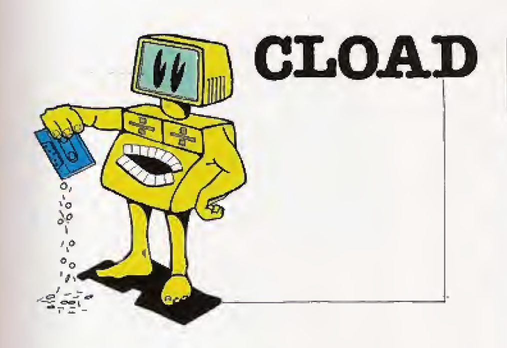
```
CLOAD"PEPE"
CLOAD"NOMBRE"
CLOAD""
```

La forma de trabajar con este comando es muy simple. Una vez introducido a través del teclado y tras accionar la tecla RETURN (retorno de carro), el ordenador pasará a examinar las señales procedentes de la unidad de casete. En el caso de que la señal analizada coincida con el programa que se desea cargar, procederá a su lectura y a su transferencia a la memoria interna. Para evitar errores durante la operación es conveniente situar la cinta lo más cerca posible del punto en el que comienza el programa. Para ello se hace necesario disponer de alguna referencia que per-





Por efecto de la orden SAVE la información contenida en la memoria central del ordenador fluye hacia la unidad de almacenamiento externo; esta última se encarga de grabarla en el soporte adecuado.



CLOAD

Carga en la memoria interna del ordenador un programa almacenado en casete.

Formato: CLOAD "[<*nombre del programa*>]"[,*R*]

Ejemplos: CLOAD "PROG"
CLOAD ""
CLOAD "TEMA",R

mita identificar el punto en cuestión; por ejemplo, bastará con apoyarse en el estado del cuentavueltas que acostumbra a equipar a las unidades de casete.

Como ya se indicó, algunos ordenadores no admiten el campo correspondiente al nombre dentro del comando CSAVE. Por supuesto, tampoco lo admitirán ahora como argumento de CLOAD. En tal caso, la carga se ordenará de forma análoga a cuando se desconoce el nombre otorgado al fichero: CLOAD"".

Verificación de los programas almacenados

Las unidades de casete no están exentas de problemas; unas veces porque falla la alimentación en un cierto instante, otras porque las cabezas lectoras o grabadoras están sucias, e incluso, en ocasiones, por efecto de interferencias originadas por señales de radio o por algún aparato eléctrico conectado a la red de tensión. Todas estas circunstancias repercuten en que la calidad de las grabaciones sea, a veces, defectuosa, hasta el punto de que resulte imposible recuperar el programa grabado.

Muchos dialectos BASIC incorporan un comando especializado que permite comprobar si la grabación se ha efectuado correctamente. Este suele ser el comando CLOAD?, cuyo formato es:

CLOAD?"<nombre del programa>"

Por supuesto, el nombre del programa debe coincidir con el utilizado en el instante de su grabación.

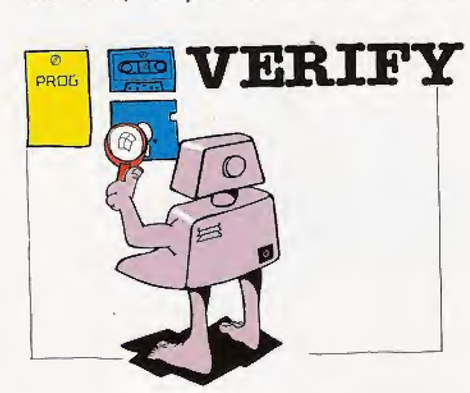
Veamos cuál es su actuación. Una vez que se ha grabado el programa con la orden CSAVE, y antes de borrar la memoria o de introducir cualquier modificación en el programa, puede entrar en escena el comando de verificación. En primer lugar, hay que rebobinar la cinta hasta el punto en el que dio comienzo la grabación; tras ello, se introducirá la orden CLOAD? acompañada por el nombre con el que fue grabado el programa a verificar. La ejecución del mismo hará que el ordenador compruebe la total coincidencia del programa que reside en su memoria con el programa del mismo nombre que se encuentra en la casete.

En algunos dialectos BASIC este comando no recibe el nombre de CLOAD?, sino que obedece al nombre de VERIFY; no obstante, su funcionamiento es análogo al descrito.

Almacenamiento en la unidad de disco

Al margen de las unidades de casete, las más extendidas son las de disco flexible. Estas últimas representan grandes ventajas, especialmente por lo que respecta al tiempo de acceso a los programas y a la velocidad de carga y grabación de los mismos.

Una notable prestación de las unidades de disco es que, de manera instantánea, es posible conocer el número de programas almacenados en el disco, así como sus respectivos nombres. Hay que tener en cuenta que las unidades de disco reservan una zona del soporte, denominada *directorio,* para almacenar dicha información. El directorio es una suerte de índice o catálogo en el que están reflejados todos y cada uno de los programas que se han ido grabando, señalando su localización exacta en el soporte. Con ello, es posible acceder directa-



CLOAD?

Verifica la coincidencia de un programa grabado en cinta con el que se encuentra en la memoria interna.

Formato: CLOAD? "<*nombre del programa*>"

Ejemplo: CLOAD? "PEP"

mente a cada uno de los programas tras una consulta al directorio.

La desventaja de estas unidades frente a las de casete reside en el precio, notablemente superior; aunque como contrapartida ofrecen unas prestaciones superiores, que permiten el empleo de pequeños ordenadores en aplicaciones (comerciales, de gestión...) que sin su colaboración quedarían vedadas.

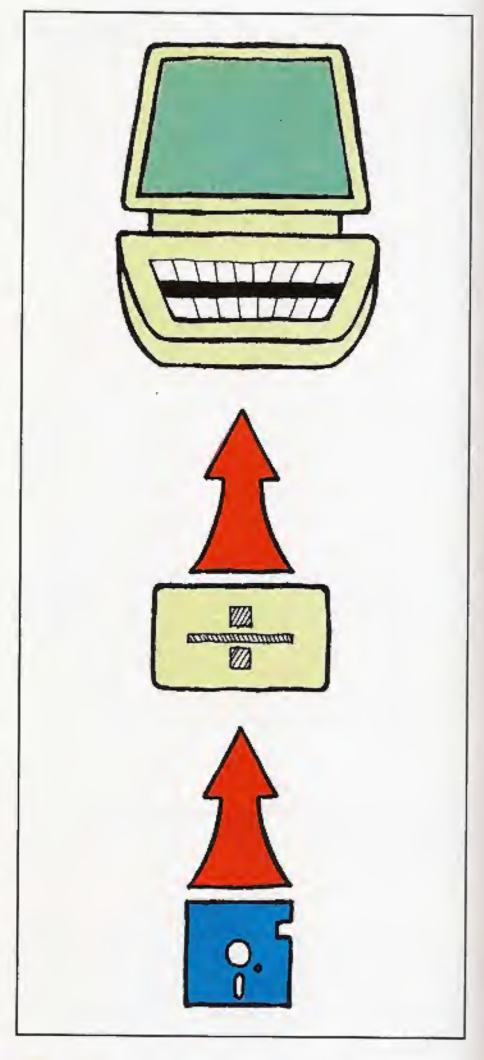
La operación básica a realizar con una unidad de disco consiste en la grabación de programas. El comando BASIC al efecto suele coincidir con SAVE, cuya formulación habitual es:

SAVE <periférico>:<nombre del programa>

Realmente, SAVE tiene en muchos ordenadores la doble utilidad de ordenar el almacenamiento de información tanto en casete como en disco. Este es, precisamente, el motivo por el que se incluye la opción <periférico> en su argumento; ésta debe referenciar a la unidad en la que hay que grabar la información. La opción a utilizar para que el programa sea enviado a la unidad de casete es CAS, aunque, normalmente, también será éste el destino del programa si no se especifica referencia alguna. Para elegir la unidad de disco como destinataria de la grabación hay que colocar en dicho campo el número de la unidad implicada (1,2, ...). El nombre del programa ha de ajustarse a unas condiciones semejantes a las que se imponían en el caso del casete; esto es: no debe sobrepasar una determinada longitud (longitud que varía de uno a otro ordenador) y puede admitir o no «apellidos», dependiendo de la máquina en cuestión. Los siguientes ejemplos son formulaciones habituales del comando SAVE:

```
SAVE "CAS:LOC1"
SAVE "1:GUSA"
SAVE "2:CASA"
SAVE "PROG"
```

Cuando se trabaja con una unidad de casete, el usuario se ve obligado normalmente a realizar las funciones de controlador de cinta; en tal caso, el ordenador trabaja «a ciegas» grabando o leyendo de la zona en la que esté posicionado el cabezal, y sin comprobar si en ese lugar había algo previamente grabado. Desde luego, ello significa que en una misma cinta es posible grabar dos programas con el mismo nombre. Esta característica no es extensiva a la unidad de disco. Ahora, el ordenador es el encargado de controlar el disco, auxiliándose para ello en el directorio. En consecuencia, si se ordena la escritura de un programa cuyo nombre coincide con el de otro programa presente en el disco, el ordenador grabará dicho programa encima del ya existente. Dado que el control de la unidad de disco es automático, el trabajo del operador se limita a comunicar a la máquina las instrucciones adecuadas.



Al ejecutar una instrucción LOAD, la información almacenada en el soporte externo es leída por el ordenador y depositada en la memoria central.

SAVE

Graba el programa que se encuentra en la memoria del ordenador en una unidad de almacenamiento externo (disco o casete).

Formato: SAVE "[*<periférico de destino>*:][*<nombre del programa>*]"

Ejemplos: SAVE "CAS : GAME"
SAVE "1 : CATO"

Lectura de programas en disco

Para recuperar los programas de la unidad de disco es necesario recurrir al comando LOAD, cuya formulación general coincide con:

TABLA DE CONVERSION							
ORDENADOR	**CLOAD**	**CSAVE**	**Verificación**	**LOAD**	**SAVE**	**Autoejecución**	
	CLOAD "<nom.>"	**CSAVE "<nom.>"**	**CLOAD? "<nom.>"**	**LOAD"<per.>: "<nom.>"**	**SAVE"<per.>: "<nom.>"**	**CSAVE "<nom.>",AUTO**	**LOAD"<per.>: "<nom.>",R**
AMSTRAD	LOAD "<nom.>"	save "<nom.>"	—	LOAD "<per.>: <nom.>"	SAVE "<per.>: <nom.>"	—	RUN "<per.>: <nom.>"
APPLE II (APPLESOFT)	—	—	VERIFY "[nom.]"	LOAD "[nom.]"	SAVE "[nom.]"	—	RUN [nom.]
APRICOT (M-BASIC)	—	—	—	LOAD "<nom.>"	SAVE "<nom.>"	—	LOAD "<per.>: <nom.>",R RUN "<nom.>"[<,R>]
ATARI	CLOAD	CSAVE	—	LOAD "<per.>: <nom.>"	SAVE "<per.>: <nom.>"	—	—
CBM 64	—	—	VERIFY ["<nom.>"] [,<per.>]	LOAD ["<nom.>"] [,<per.>]	SAVE ["<nom.>"] [,<per.>]	—	(1)
DRAGON	CLOAD "<nom.>"	CSAVE "<nom.>"	—	—	—	—	—
EQUIPOS MSX	CLOAD ["<nom.>"]	CSAVE "<nom.>"	CLOAD? ["<nom.>"]	LOAD "<PER>: <nom.>"	SAVE "<per>: <nom.>"	—	LOAD"<per.>: <nom.>",R
HP-150	—	—	—	LOAD "<nom.>"	SAVE "<nom.>"	—	LOAD "<nom.>",R
IBM PC	—	—	—	LOAD "<per.>: <nom.>"	SAVE "<per.>: <nom.>"	—	LOAD "<per.>: <nom.>",R
MPF	LOADT "<nom.>"	SAVET "<nom.>"	—	—	—	—	—
NCR DM-V (MS-BASIC)	CLOAD "<nom.>"	CSAVE "<nom.>"	CLOAD? "<nom.>"	LOAD "<nom.>"	SAVE "<nom.>"	—	LOAD "<per.>: <nom.>", R
NEW BRAIN	LOAD	SAVE "<nom.>"	VERIFY	—	—	—	—
ORIC	CLOAD "[<nom.>]"	CSAVE "<nom.>"	CLOAD ["<nom.>"],V	—	—	CSAVE "<nom.>", AUTO	—
SHARP MZ-700 (MZ-BASIC)	LOAD "[<nom.>]"	SAVE <per.>	VERIFY "[<nom.>]"	—	—	—	—
SINCLAIR QL	CLOAD "<nom.>"	—	—	LOAD ["]<per.> <nom.>["]	SAVE["]<per.> <nom.>["]	—	—
SPECTRA-VIDEO	CLOAD "<nom.>"	CSAVE "<nom.>"	CLOAD? "<nom.>"	LOAD "<per.>: <nom.>"	SAVE "<per.>: <nom.>"	—	LOAD "<per.>: <nom.>",R
ZX-SPECTRUM	LOAD "[<nom.>]"	SAVE <per.>	VERIFY "[<nom.>]"	—	—	SAVE["<nom.>]" LINE<n>	—

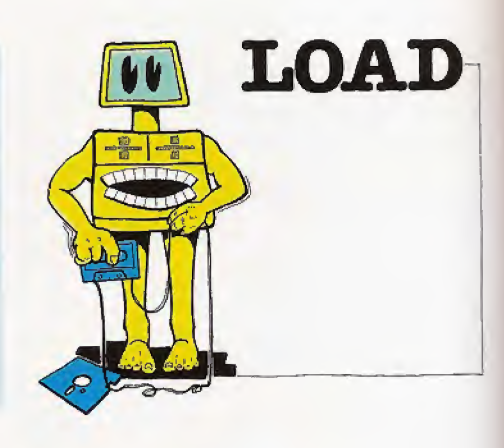
<nom.>: Nombre del programa con el que se trabaja. <per.>: Especifica el periférico a emplear.

FORMULACIONES DE LOS COMANDOS

CLOAD <nom.>: Carga en memoria el programa almacenado en casete cuyo nombre coincide con el especificado. CSAVE <nom.>: Comprueba la total coincidencia del programa cargado en memoria con el residente en la unidad de casete. LOAD "<per.>: <nom.>": Carga en memoria el programa indicado que reside en la unidad de almacenamiento externo que se especifica en la zona <per.>. Si se omite la opción <per.>, la unidad implicada será el disco. SAVE "<per.>: <nom.>": Almacena en el periférico indicado el programa almacenado en memoria, otorgándole el nombre señalado. Cuando la elección de periférico no existe, la grabación se realizará en la unidad de disco. CSAVE "<nom.>"AUTO: Graba en casete el programa indicado, acompañado de un indicativo que hará que el ordenador lo ejecute automáticamente en el momento de cargarlo en la memoria interna. LOAD "<per.>: <nom.>",R: Formulación del comando LOAD que ordena la ejecución del programa una vez concluida la carga en memoria.

OBSERVACIONES:

(1) En el CBM 64, la instrucción LOAD como parte de un programa implica la ejecución de dicho programa.

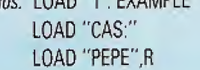


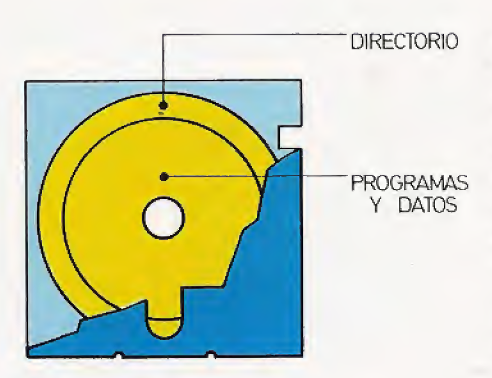
LOAD

Carga en la memoria del ordenador un programa residente en una unidad de almacenamiento externo. La opción ",R" ordena su autoejecución automática.

Formato: LOAD "[<*periférico*>:][<*nombre del programa*>]"[,*R*]

Ejemplos: LOAD "1 : EXAMPLE"
LOAD "CAS:"
LOAD "PEPE",R





Las unidades de disco permiten conocer, de forma casi instantánea, cuál es el número y nombre de los programas almacenados en cada disco magnético. Ello es posible debido a que las referidas unidades reservan una zona del disco, denominada «directorio», para el almacenamiento de dicha información.

LOAD"<periférico><nombre del programa>",R

Las opciones son las mismas que las del comando SAVE, de tal forma que es necesario especificar el periférico y el nombre del programa. La R que aparece como campo final, separado por una coma, sirve para indicar a la máquina que el programa debe ser ejecutado automáticamente al concluir la operación de carga. Su presencia sustituye virtualmente al comando RUN. Dos ejemplos de formulación correcta son:

```
LOAD "CAS:CARP"
LOAD "2:MIO"
```

Como observación final, cabe añadir que al realizar la lectura y transferencia del programa desde la unidad de disco a la memoria central, quedará borrado el contenido previo de esta última.

Autoejecución de programas

En ciertos casos, es necesario que un programa que se encuentra en una unidad de almacenamiento externo, se atoejecute inmediatamente tras ser cargado en la memoria central. La necesidad puede obedecer a distintas causas. Por ejemplo, puede ocurrir que, por algún motivo, sea necesario encadenar programas (el programa en curso llama a otro que se encuentra en la unidad de almacenamiento). En tal situación, es necesario, o cuando menos conveniente, que el programa llamado se ejecute automáticamente, sin necesidad de que el operador intervenga en el proceso. Esta opción se utiliza con frecuencia para proteger programas profesionales, de tal forma que la ejecución del referido programa no pueda se detenida para analizar su codificación y, posiblemente, copiarlo.

Una de las posibilidades para ordenar la autoejecución reside en la opción "R" que puede acompañar al comando LOAD; por ejemplo:

```
LOAD "CAS:PEPE",R
LOAD "1:PROGRAM",R
```

El tradicional comando de ejecución, RUN, también es utilizable en ciertos dialectos BASIC para este cometido. En tal caso, admite una formulación distinta a la ya conocida. Esta es:

RUN"<periférico>:<nombre>"

Al ejecutarla se consigue el mismo resultado que con LOAD y la opción R. Hay que precisar que la opción <periférico> debe figurar tal y como se indicó en un apartado precedente. Por lo demás, el nombre corresponde al otorgado al programa a ejecutar. Por ejemplo:

```
RUN "CAS:PEPE"
RUN "2:PROG8"
```

LOAD?

Verifica coincidencia del programa cargado en la memoria interna con el que se encuentra en la unidad de almacenamiento.

Formato: LOAD? "[<*periférico*>:][<*nombre del programa*>]

Ejemplo: LOAD? "CAS: BUT"

Toma de decisiones

Rupturas de secuencia condicionales e incondicionales

Un programa es, en esencia, un conjunto de instrucciones que detallan un trabajo a realizar por el ordenador. La perfecta conclusión del trabajo programado se obtiene al ejecutar ordenadamente las diversas instrucciones; esto es: en primer lugar, la máquina ejecutará la primera línea del programa, a continuación la segunda y así sucesivamente hasta terminar.

En algunos casos, es conveniente que el programa no se ejecute de forma totalmente secuencial. Hay ocasiones en las que interesa que tras una cierta instrucción no se ejecute la que está precedida por el siguiente número de línea, sino otra localizada en otro punto del programa. Para facilitar esta posibilidad, el BASIC ofrece medios adecuados para la *ruptura de secuencia.*

La instrucción GOTO

La ruptura de la secuencia de ejecución de un programa puede ser, desde luego, obligatoria (incondicional); no obstante, también es posible que la ruptura no deba realizarse en cualquier caso, sino que dependa del cumplimiento de una condición impuesta. Esta distinción está contemplada en el lenguaje BASIC y de ahí que existan instrucciones distintas que permiten la programación tanto de rupturas *incondicionales* como *condicionales.* GOTO es el comando BASIC que permite construir las instrucciones para la ruptura de secuencia o bifurcación incondicional. Su formato general es:

GOTO <número de línea>.

El número de línea que figura como argumento corresponderá a la instrucción que deseamos se ejecute a continuación; esto es: a la instrucción a la que debe realizarse el salto o bifurcación. Una vez ejecutada ésta, el programa seguirá ejecutándose secuencialmente a partir del referido número de línea.

Muchas versiones del lenguaje BASIC permiten incluir dentro del comando GOTO una expresión matemática; una vez calculada, su resultado identificará el número de línea al que hay que saltar. Por ejemplo, las dos instrucciones que siguen son equivalentes:

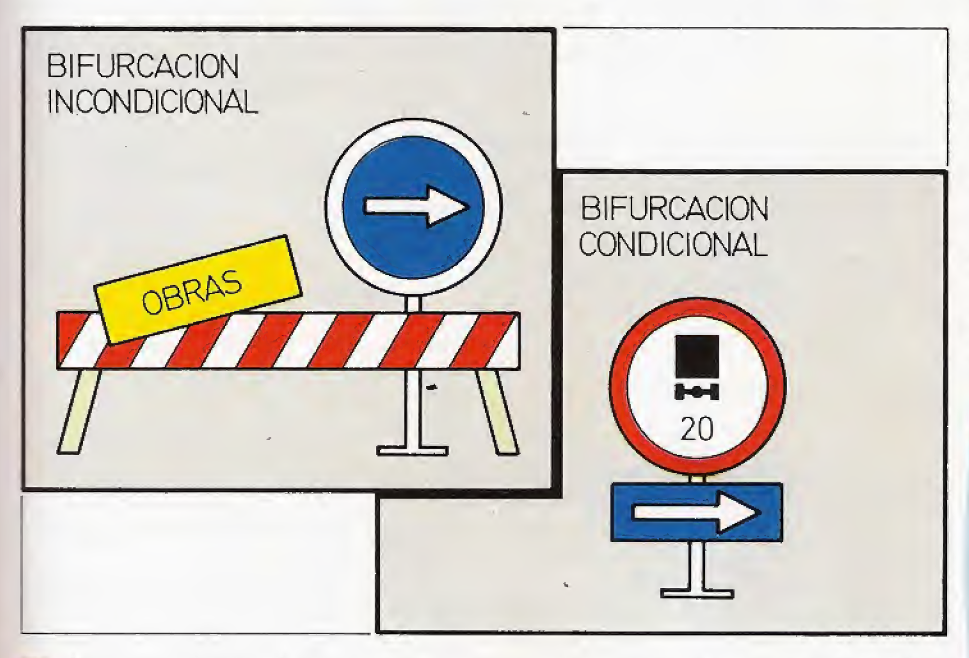
```
GOTO 100
GOTO 50*2
```

Ambas instrucciones ordenan un salto a la línea 100 del programa. La utilidad de la instrucción GOTO se manifiesta especialmente a la hora de programar tareas repetitivas. Por ejemplo, el siguiente programa recurre a una instrucción GOTO para realizar el cálculo de los cuadrados de los sucesivos números enteros (1, 2, 3...). El programa constituye un bucle sin salida; el usuario debe interrumpir su ejecución por medios directos.

```
10 REM CUADRADOS
20 LET N=1
30 PRINT "EL CUADRADO DE";N;"ES:" N*N
40 LET N=N+1
50 GOTO 30
█
```

El programa resulta de lo más simple. La línea 20 da a la variable N el valor inicial uno. Acto seguido está localizada la primera instrucción del bucle repetitivo. Su cometido es mostrar el texto "EL CUADRADO DE", seguido por el valor del número (N) y el texto siguiente ("ES:"); terminando con el cálculo e impresión del cuadrado del número en cuestión (N*N). La línea 40 se encarga de incrementar en una unidad el valor de N; de esta forma, al regresar a la instrucción 30, por efecto del GOTO 30 localizado en la línea siguiente, se procederá al cálculo y presentación en pantalla del cuadrado del siguiente número.

```
RUN
EL CUADRADO DE 1 ES: 1
EL CUADRADO DE 2 ES: 4
EL CUADRADO DE 3 ES: 9
EL CUADRADO DE 4 ES: 16
EL CUADRADO DE 5 ES: 25
EL CUADRADO DE 6 ES: 36
EL CUADRADO DE 7 ES: 49
EL CUADRADO DE 8 ES: 64
```



El lenguaje BASIC ofrece al programador comandos adecuados para ordenar saltos o bifurcaciones incondicionales (GOTO) y condicionales (IF/THEN); estos últimos se apoyan en el cumplimiento o no de una o varias condiciones impuestas.

Toma de decisiones

Uno de los aspectos que diferencian más notoriamente a los ordenadores del resto de la extensa gama de máquinas electrónicas de cálculo (registradoras, calculadoras de bolsillo, etc...) es, sin duda alguna, la posibilidad de tomar decisiones. A partir de datos suministrados, ya sea por el programa en curso de ejecución, o bien por el usuario que esté en ese preciso instante frente al teclado de la máquina, el ordenador es capaz de realizar una evaluación de acuerdo a lo consignado en el programa y tomar las decisiones para las que está instruido. No obstante, hay que partir del hecho de que el ordenador por sí mismo no tiene poder de decisión alguno; debe ser el propio programador quien establezca los términos de la condición a verificar, así como la acción o acciones que debe emprender la máquina como respuesta. Por ejemplo, suponga que se trata de dar al ordenador una serie de cinco números diferentes, para que los sume y presente en la pantalla el resultado. Una primera solución programada podría ser la siguiente:

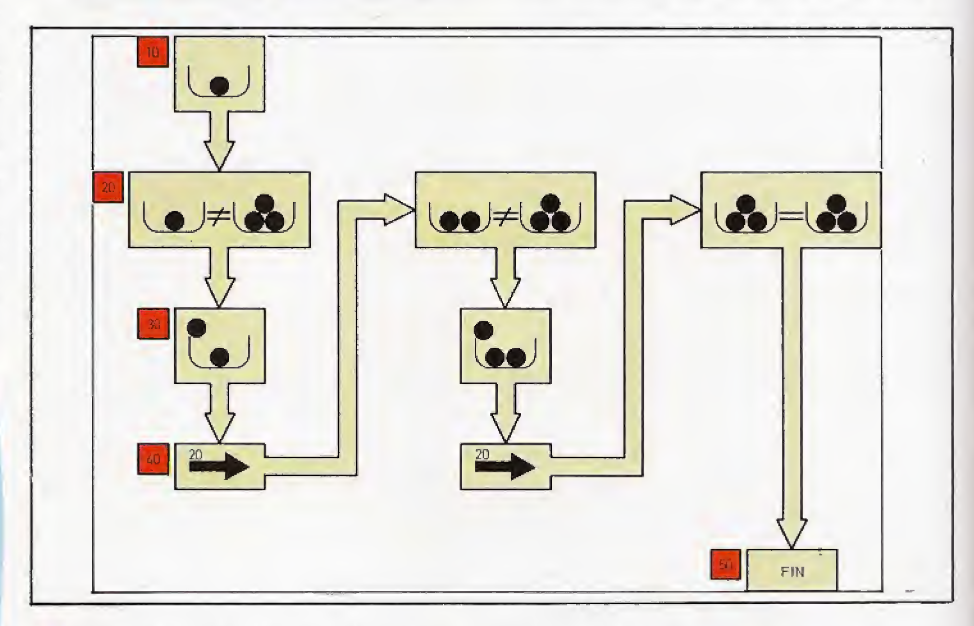
```
10 LET B=0
20 PRINT "INTRODUZCA UN NUMERO"
30 INPUT A
40 LET B=B+A
50 GOTO 20
█
```

El programa pide la introducción de los sucesivos datos, calcula la suma y la presenta en la pantalla; sin embargo, no está capacitado para detectar cuándo se le han dado los cinco números. La ejecución cierra un bloque continuo, sin detención, excepto por medios ajenos al propio programa.

Es preciso contar con un mecanismo que permita instruir al ordenador para que sea capaz de tomar decisiones. De esta forma, será posible construir un programa que detecte la introducción de los cinco valores y, en ese instante, «decida» presentar la suma de ellos y detener automáticamente el proceso en ejecución.

```
IF/THEN
10 LET N=1
20 IF N=3 THEN 50
30 LET N=N+1
40 GOTO 20
50 END
```

Las instrucciones IF/THEN/ELSE (en su fórmula completa u omitiendo la zona ELSE) permiten programar tomas de decisión asociadas a bifurcaciones o saltos en la secuencia de ejecución del programa.



La figura ilustra la ejecución del programa que aparece en una pantalla anterior, y que incluye los dos tipos de saltos: condicional (línea 20) e incondicional (línea 40). Tal como se observa, el programa ejecuta una acción reiterada (acumulación de sucesivos elementos) hasta que se cumple la condición establecida en la instrucción 20 (N=3), momento en el cual se produce un salto a la instrucción final.

Estructuras de control

Las herramientas de decisión que aporta el lenguaje BASIC se plasman en un conjunto de instrucciones que cabría denominar *estructuras de control*. Las estructuras de control alteran el flujo o la secuencia de ejecución del programa, haciendo que en un determinado momento se ejecuten un conjunto de instrucciones u otro, según se verifique o no una determinada condición.

Existen diversos tipos de estructuras de control, si bien, la principal en el BASIC es la IF/THEN/ELSE.

Su traducción a lenguaje convencional ilustra claramente sus posibilidades: IF (Si) se cumple la condición, THEN (Entonces) ejecutar una determinada instrucción, ELSE (De lo contrario) ejecutar la instrucción siguiente del programa.

Otra estructura de control, muy útil a la hora de tomar decisiones, es WHILE/DO: WHILE (Mientras) se verifique la condición, DO (Hacer) ejecutar la instrucción indicada.

En este caso, la condición que debe verificarse para que se ejecute la acción asociada al DO, se evalúa antes de ejecutar por vez primera la instrucción o acción especificada. Esto significa que si al llegar a una estructura WHILE/DO la

condición es falsa, no se ejecutará ninguna vez la instrucción que ocupa la zona DO.

Hay otra estructura de control muy parecida a la anterior aunque con una ligera diferencia. Esta estructura es REPEAT/UNTIL: REPEAT (Repetir) la ejecución de la instrucción indicada UNTIL (Hasta que) se verifique la condición.

El pequeño matiz que la diferencia de la estructura anterior es que, en este caso, primero se ejecuta la instrucción que sigue a la zona REPEAT y, a continuación, se verifica si se cumple o no la condición impuesta. De verificarse, se ejecutará de nuevo la instrucción, y así sucesivamente. En consecuencia, siempre se ejecuta al menos una vez la instrucción asociada a la estructura de control.

Por último, otra estructura de control bastante utilizada en multitud de programas es CASE/OF. Su utilidad se manifiesta a la hora de evaluar una condición y como consecuencia, bifurca a la instrucción asociada al estado de la condición. En definitiva, permite adoptar una decisión múltiple y ejecutar la acción o respuesta pertinente en cada caso. Las instrucciones a ejecutar se colocan ordenadamente detrás de la zona OF. Al evaluar la expresión asociada a CASE, ha de obtenerse un resultado coincidente con un número entero (1,2,3,4,...). Según sea tal valor, la instrucción OF ejecutada será la que ocupe el primer lugar, el segundo, tercero...

...Y a tomar decisiones

Aunque hay dialectos del lenguaje BASIC que incorporan en su repertorio los comandos WHILE/DO y REPEAT/UNTIL, lo habitual es que omitan tales comandos y reduzcan su oferta a la instrucción IF/THEN/ELSE. En cualquier caso, las dos estructuras de control mencionadas pueden simularse mediante el empleo adecuado de la instrucción de salto condicional IF/THEN/ELSE. Esta instrucción es una transposición práctica de la respectiva estructura de control. Por lo demás, y salvo muy pocas excepciones, esta instrucción adopta idéntico formato en casi todos los ordenadores personales presentes en el mercado.

La diferencia más generalizada se encuentra en los dialectos BASIC que omiten la zona ELSE. En todo caso, esta opción queda implícita en la propia estructura IF/THEN: si la condición que se ha de verificar no es cierta, se ejecutará la línea siguiente a la que ocupa la propia instrucción IF/THEN; de ahí que la función ELSE pueda obviarse sin mayores problemas. Incluso aunque esté disponible, el uso de la zona ELSE es opcional, de forma que en la práctica suele omitirse en muchos casos.

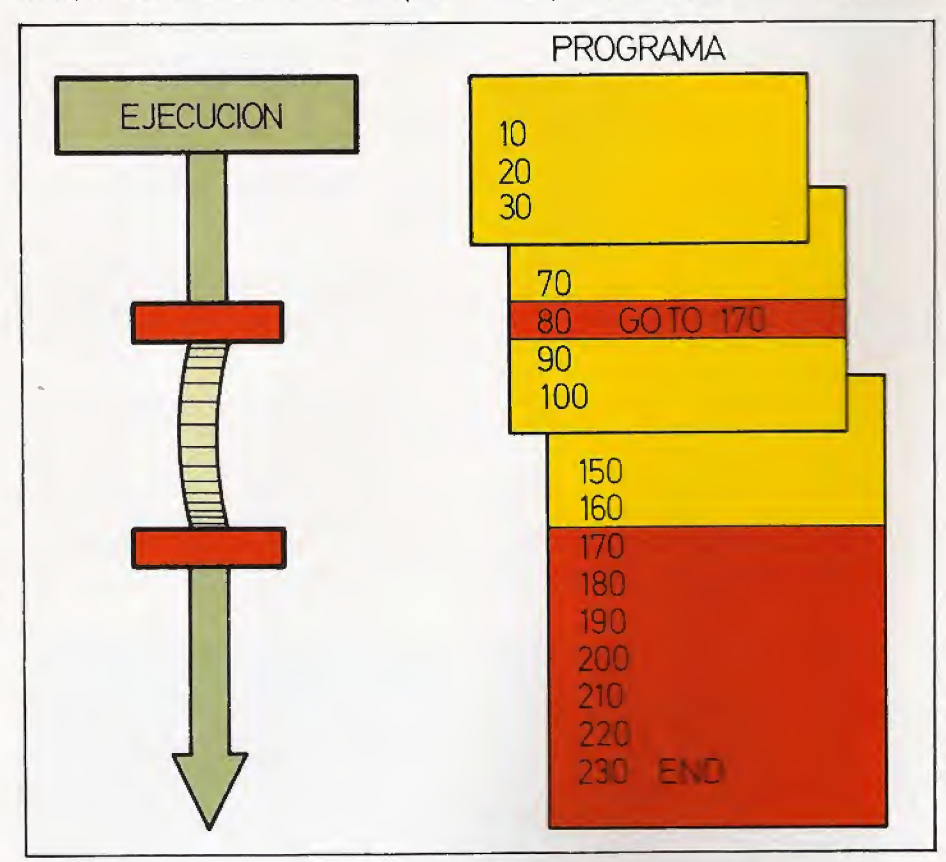
Volviendo al ejemplo anterior, ahora el ordenador sí puede estar ya en condiciones para saber cuándo se le han dado los cinco números; es posible programar la toma de decisión adecuada. Así pues, el ordenador podrá evaluar si se han introducido ya los cinco números y, en consecuencia, calcular la

GOTO

Ejecuta un salto al número de línea especificado.

Formato: (Número de línea) GOTO [<número de línea>] [<expresión>]

Ejemplos: GOTO 30
20 GOTO A+30



La instrucción GOTO es la herramienta BASIC adecuada para ordenar rupturas de secuencia o bifurcaciones incondicionales. El programa de la figura se ejecutará normalmente hasta llegar a la línea 80. A partir de este punto se rompe la secuencia por efecto de la instrucción GOTO 170, que provoca un salto en la ejecución hasta esta línea.

suma, visualizar el resultado y detener la ejecución del programa. Veamos cuál será el nuevo aspecto del programa:

```
10 LET B=0
20 LET N=0
30 PRINT "INTRODUZCA UN NUMERO:"
40 INPUT A
50 LET N=N+1
60 LET B=B+A
70 IF N<>5 THEN GOTO 30 ELSE PRINT
   "LA SUMA ES=";B
80 END
█
```

En esta ocasión, la variable N lleva la cuenta de cuántos números se van introduciendo en el ordenador; su valor —inicializado a cero— se incrementa en una unidad cada vez que se teclea un nuevo dato. A su vez, la variable B —también inicializada a cero— irá totalizando el valor de la suma de los datos introducidos.

En la línea 70 se comprueba la condición: ¿valor de N distinto de 5? Si la condición se cumple, esto es, si aún no se han introducido los cinco números a sumar, se ejecutará la zona THEN y, en consecuencia, se regresará de nuevo a la ejecución de la línea 30 (GOTO 30). Por el contrario, si la condición deja de cumplirse (N ha alcanzado ya el valor 5), se ejecutará la instrucción asociada a la zona ELSE; ésta presenta en la pantalla la suma total de los cinco datos. Acto seguido, el ordenador pasará a ocuparse de la instrucción 80 que pondrá fin a la ejecución del programa.

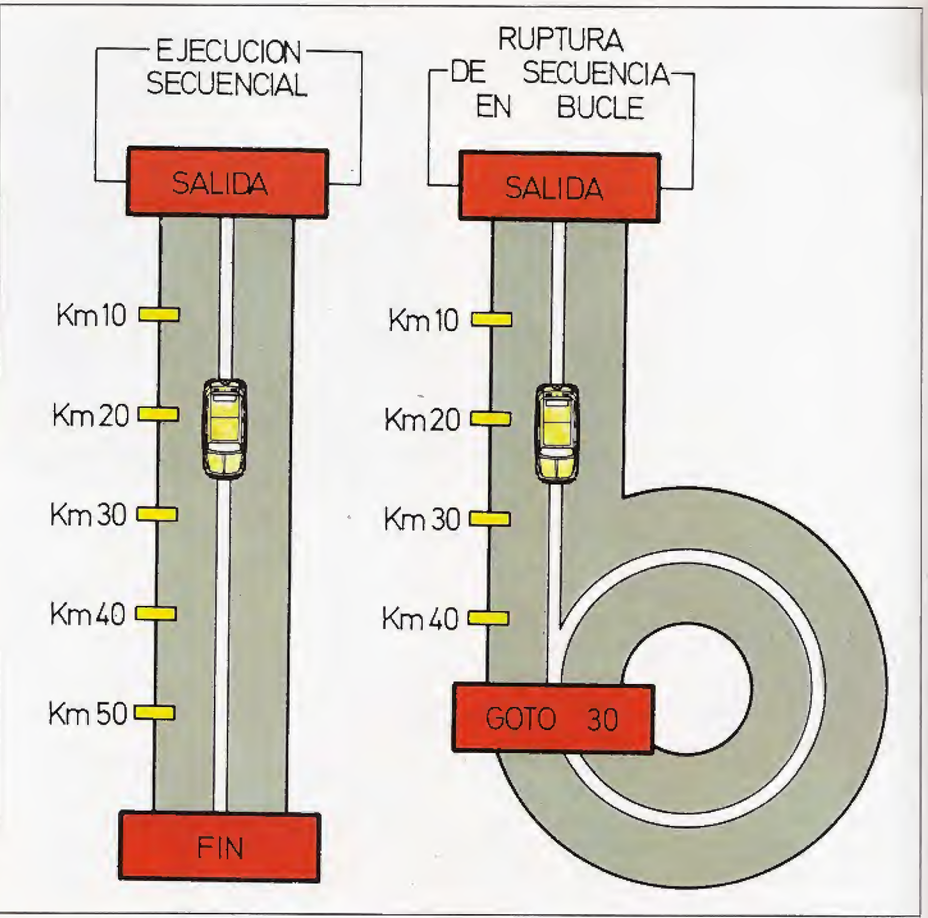
Para dotar de mayor versatilidad a esta instrucción, normalmente, es posible introducir detrás de la palabra THEN, o de la zona ELSE, más de una instrucción, separadas por el símbolo dos puntos (:). Con ello podrá ejecutarse más de una acción como respuesta a una misma decisión adoptada en la zona IF. Algunos intérpretes de BASIC permiten incluso suprimir la palabra THEN cuando ésta se completa con la instrucción GOTO. En tal caso, las dos instrucciones que siguen serán equivalentes:

```
70 IF N<>5 THEN GOTO 30
70 IF N<> GOTO 30
```

En otros dialectos BASIC, la palabra clave que se puede omitir en tal situación es GOTO. Por ejemplo:

```
70 IF N<>5 THEN GOTO 30
70 IF N<>5 THEN 30
```

De nuevo, ambas instrucciones serían equivalentes.



La posibilidad de romper la secuencia de ejecución de un programa permite al usuario programar bucles repetitivos. La presencia de la instrucción de salto hacia atrás (GOTO 30) crea un bucle iterativo que habrá que romper por medios ajenos al programa BASIC.

IF..THEN..ELSE

Ejecuta las instrucciones que siguen a THEN si se cumple la condición; en caso contrario, se ejecutan las instrucciones que siguen a ELSE.

Formato: (Número de línea) IF <condición> THEN <instrucción 1>[ELSE<instrucción 2>]

Ejemplos:
```
20 IF A>0 THEN P=1
40 IF A>0 THEN P= ELSE P=0
50 IF X=5 THEN GOTO 120
```

Estructuras de control con IF/THEN/ELSE

Una vez descrito el funcionamiento de la instrucción IF/THEN/ELSE, es posible llevarla al terreno práctico programando, por ejemplo, las restantes es-

TABLA DE CONVERSION

Ordenador	GOTO		IF/THEN/ELSE		
	GOTO <nl>	GOTO <expr.>	IF/THEN	IF/THEN/ELSE	IF {THEN / GOTO} <nl> nl
AMSTRAD	GOTO <nl>	—	IF/THEN	IF/THEN/ELSE	—
APPLE II (APPLESOFT)	GOTO <nl>	—	IF/THEN	—	IF THEN <nl>
APRICOT (M-BASIC)	GOTO <nl>	—	IF/THEN	IF/THEN/ELSE	IF {THEN / GOTO} <nl>
ATARI	GOTO <nl>	GOTO <expr.>	IF/THEN	—	IF THEN <nl>
CBM 64	GOTO <nl>	—	IF/THEN	—	IF THEN <nl>
DRAGON	GOTO <nl>	—	IF/THEN	IF/THEN/ELSE	IF THEN <nl>
EQUIPOS MSX	GOTO <nl>	—	IF/THEN	IF/THEN/ELSE	IF {THEN / GOTO} <nl>
HP-150	GOTO <nl>	—	IF/THEN	IF/THEN/ELSE	IF {THEN / GOTO} <nl>
IBM PC	GOTO <nl>	—	IF/THEN	IF/THEN/ELSE	IF GOTO <nl>
MPF	GOTO <nl>	—	IF/THEN	—	IF THEN <nl>
NCR DM-V (MS-BASIC)	GOTO <nl>	—	IF/THEN	IF/THEN/ELSE	IF {THEN / GOTO} <nl>
NEW BRAIN	GOTO <nl>	—	IF/THEN	—	IF {THEN / GOTO} <nl>
ORIC	GOTO <nl>	GOTO <expr.>	IF/THEN	IF/THEN/ELSE	—
SHARP MZ-700 (MZ-BASIC)	GOTO <nl>	—	IF/THEN	—	IF {THEN / GOTO} <nl>
SINCLAIR QL	GOTO <nl>	GOTO <expr.>	IF/THEN	IF/THEN/ELSE	—
SPECTRAVIDEO	GOTO <nl>	—	IF/THEN	IF/THEN/ELSE	IF {THEN / GOTO} <nl>
ZX-SPECTRUM	GOTO <nl>	GOTO <expr.>	IF/THEN	—	—

<nl>: Número de línea. <expr.>: Expresión con datos y/o variables.

FORMULACIONES DE LOS COMANDOS

GOTO <nl>: Salto incondicional a la línea cuyo número se indica. GOTO <expr.>: Salto incondicional a la línea cuyo número es el resultado de la expresión. IF/THEN: Bifurcación condicional. Ejecuta la instrucción o instrucciones asociadas a THEN si se cumple la condición expresada en la zona IF. IF/THEN/ELSE: Bifurcación condicional con zona ELSE. La instrucción o instrucciones asociadas a ELSE se ejecutarán en el caso de no verificarse la condición impuesta en IF.

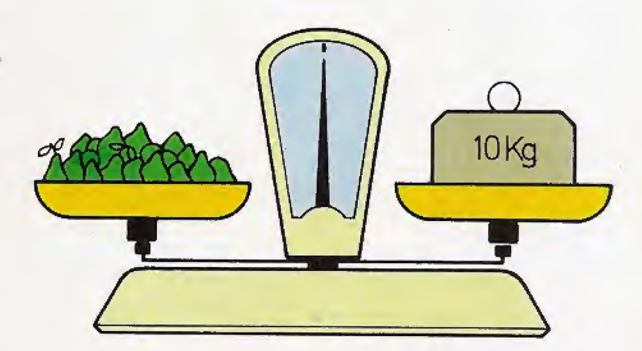
IF {THEN / GOTO} <nl>: Posibilidad de omitir una de las palabras clave (THEN o GOTO) cuando la zona THEN contiene una instrucción incondicional.

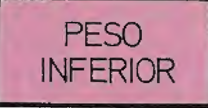



```
 5 REM TOMA DE DECISION
10 INPUT "¿CUAL ES EL COLOR?"; C$
20 IF C$="VERDE" THEN GOTO 40 ELSE PRINT
   "INTENTELO DE NUEVO"
30 GOTO 10
40 PRINT "¡BRAVO, LO ACERTO!"
50 END
```

tructuras de control a partir de la propia IF/THEN/ELSE. El ejemplo que sigue simula totalmente el comportamiento de la estructura de decisión WHILE/DO:

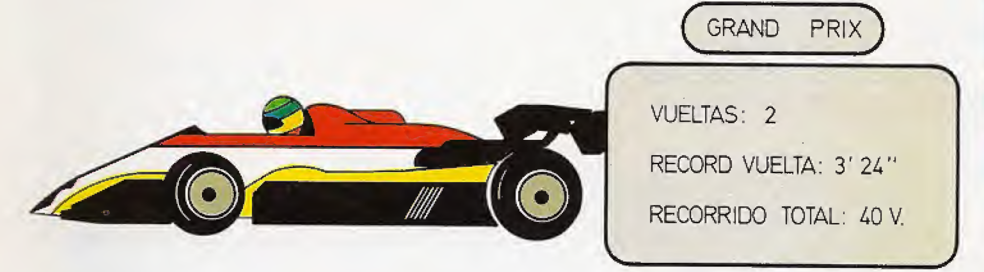
Puesta en práctica de una toma de decisión por medio de una instrucción del tipo IF/THEN/ELSE. El programa formula una pregunta al usuario y comprueba si la respuesta es o no correcta (línea 20). En el primer caso presentará el mensaje oportuno antes de finalizar el programa. Si la respuesta es incorrecta, la zona ELSE lo comunicará al usuario y la instrucción GOTO 10 ocasionará una nueva ejecución del programa.



WHILE 

DO
AÑADIR PERAS

Otra estructura de control muy frecuente es WHILE/DO: «mientras se verifique la condición impuesta (peso inferior), ejecutar la acción indicada (añadir peras a la balanza)».



REPEAT 

UNTIL
FIN DEL RECORRIDO

Estructura de control REPEAT/UNTIL: «repetir la acción indicada (dar vueltas al circuito) hasta que se verifique la condición (fin de las cuarenta vueltas que constituyen el recorrido)».

```
10 INPUT "INTRODUZCA EL NUMERO:";A
20 IF A=0 THEN GOTO 50
30 PRINT A,2*A,A*A
40 GOTO 10
50 END
```

La actividad del programa consiste, sencillamente, en presentar en la pantalla el número introducido, el doble de su valor y el cuadrado del número en cuestión. Esta tarea se lleva a cabo siempre que el número tecleado sea distinto de cero. Si el número que se introduce es el cero, la condición será cierta y, en consecuencia, el programa concluirá por efecto del GOTO 50 emplazado en la zona THEN.

Cabe observar que si el primer número introducido es el cero, no se ejecutarán ninguna vez las líneas 30 y 40; como se recordará, ésta es una característica inherente a la estructura de control WHILE/DO.

La estructura REPEAT/UNTIL se obtiene fácilmente por medio de una instrucción IF/THEN/ELSE. Una demostración elocuente la brinda el siguiente ejemplo, cuya funcionalidad es análoga a la del programa anterior:

```
10 INPUT "INTRODUZCA EL NUMERO:"; A
20 PRINT A,2*A,A*A
30 IF A<>0 THEN GOTO 10
40 END
```

En este caso, aunque el primer número que se introduzca a través del teclado sea el cero, se mostrará en la pantalla el resultado de las operaciones realizadas. Por lo tanto, siempre se ejecutará, al menos una vez, la acción que corresponde a la zona REPEAT; característica ésta peculiar de la estructura de control REPEAT/UNTIL.

Operadores de relación

Distintos métodos para comparar datos



as expresiones que se utilizan habitualmente para imponer una condición suelen ser comparaciones entre datos. En la mayoría de los casos interesa ejecutar órdenes distintas en función del valor de una determinada variable. Por ejemplo: calcular la raíz cuadrada de una variable sólo cuando ésta almacena un valor positivo y eludir el cálculo si el valor es negativo:

```
IF A>0 THEN RAIZ=SQR (A)
```

Es evidente, pues, que se hace necesario contar con un método que permita formular adecuadamente las comparaciones entre diferentes datos.

Esta comparación se codifica en BASIC por medio de los siguientes operadores de relación:

```
=, <, >, < >, <=, >=
```

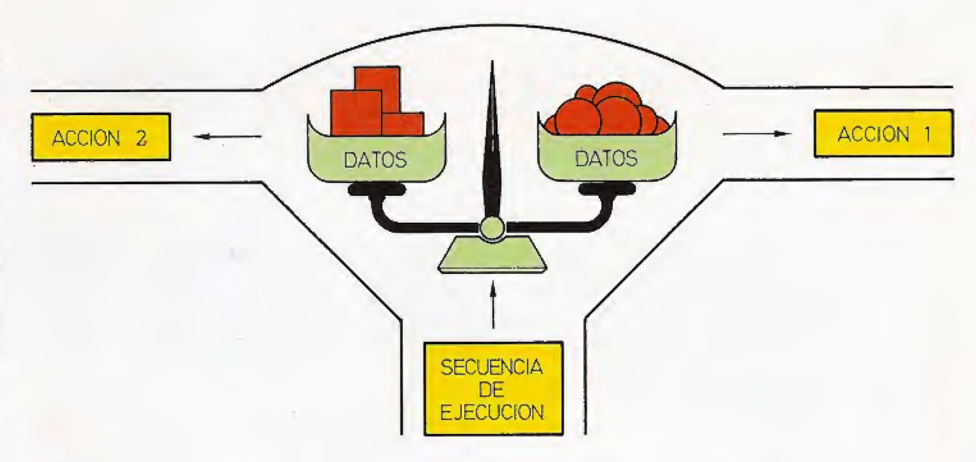
Los operadores de relación se aplican entre dos datos, constantes o variables, dando como resultado un valor de tipo lógico, esto es: respondiendo con *cierto* o *falso*.

El operador «=» evalúa la igualdad entre los datos especificados, de tal forma que si son iguales la expresión adoptará el valor *cierto*, y si son distintos el valor *falso*.

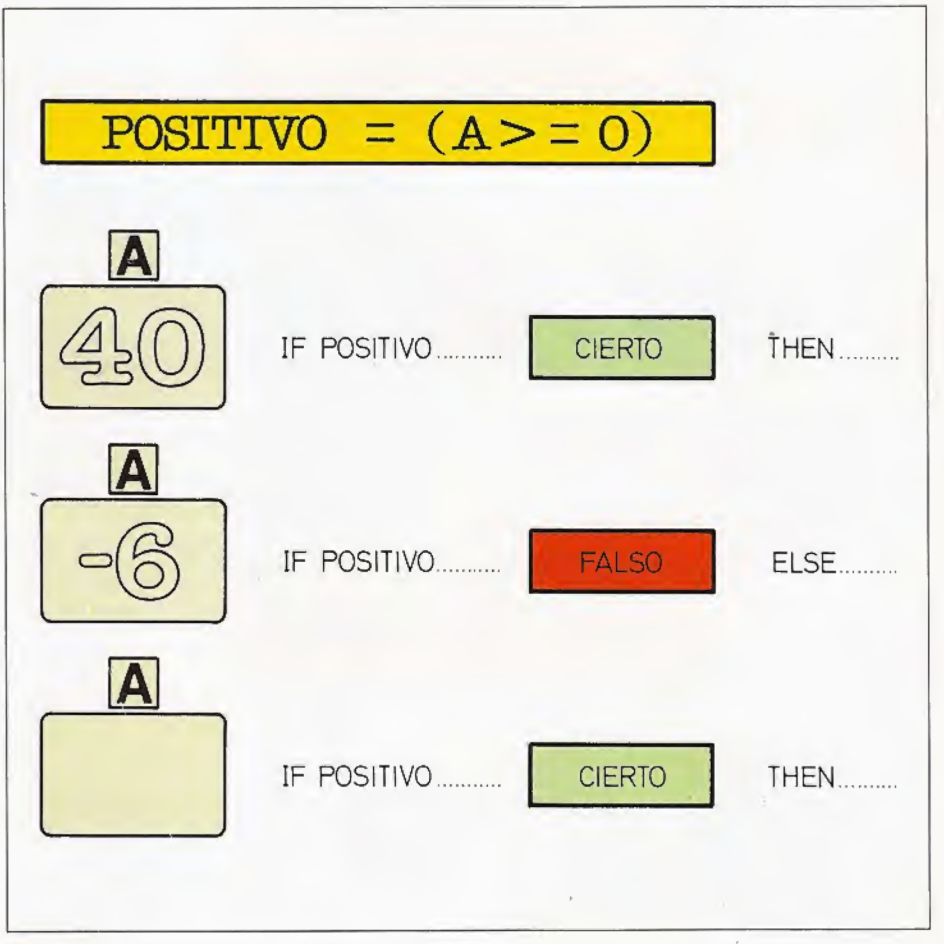
```
3=3 ...CIERTO
3=4 ...FALSO
5=2 ...FALSO
```

El operador «<» asignará el valor *cierto* a la expresión en la que se encuentre cuando el dato situado a su izquierda sea estrictamente menor que el dato colocado a su derecha. Si el valor de la izquierda es mayor o igual que el valor de la derecha, la expresión tomará el valor *falso*.

```
3<4 ...CIERTO
3<3 ...FALSO
5<2 ...FALSO
```

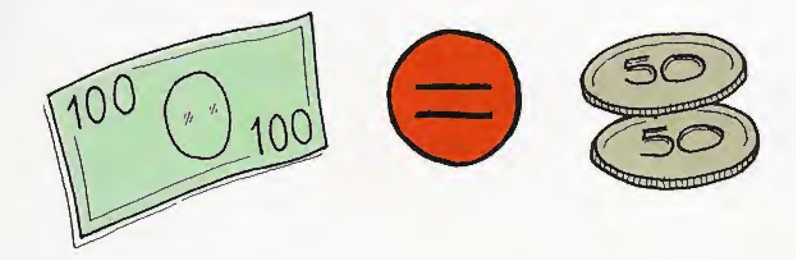


La secuencia de ejecución de los programas puede sufrir alteraciones de acuerdo con el cumplimiento o no de ciertas condiciones impuestas por el programador. Estas suelen adoptar la forma de comparaciones entre datos establecidas por medio de operadores de relación.



El resultado de evaluar una condición es un valor lógico: «-1» para cierto y «0» para falso. Este valor puede almacenarse en una variable numérica y ser utilizado dentro de instrucciones de bifurcación condicional. En el ejemplo, la variable POSITIVO —cuyo valor queda establecido en la asignación inicial— se utiliza para imponer la condición de la instrucción IF/THEN/ELSE.

IGUAL QUE



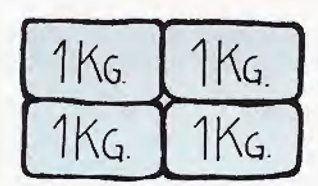
El operador «igual que» (=) evalúa la igualdad entre los dos datos colocados a los lados del operador.

MENOR QUE



La condición «menor» (<) responde con «cierto» (1 lógico) cuando el dato situado a la izquierda del mismo es inferior al situado a su derecha.

MAYOR QUE







La tercera de las condiciones básicas se concreta en la relación «mayor que» (>), cierta si el dato situado a la izquierda del operador es de magnitud superior al de la derecha.

El signo «>» indica «mayor que»; es análogo al anterior, si bien, en este caso, el operando de la izquierda debe ser mayor que el de la derecha para que el valor obtenido sea *cierto.*

3>3 ...FALSO
3>4 ...FALSO
5>2 ...CIERTO

Los operadores definidos se complementan con otros tres que realizan funciones contrarias.

El operador complementario de la igualdad es el de desigualdad, cuyo signo es «<>». Cuando los datos colocados a izquierda y derecha del referido signo sean distintos, el resultado de la comparación será *cierto.*

3<>3 ...FALSO
3<>4 ...CIERTO
5<>2 ...CIERTO

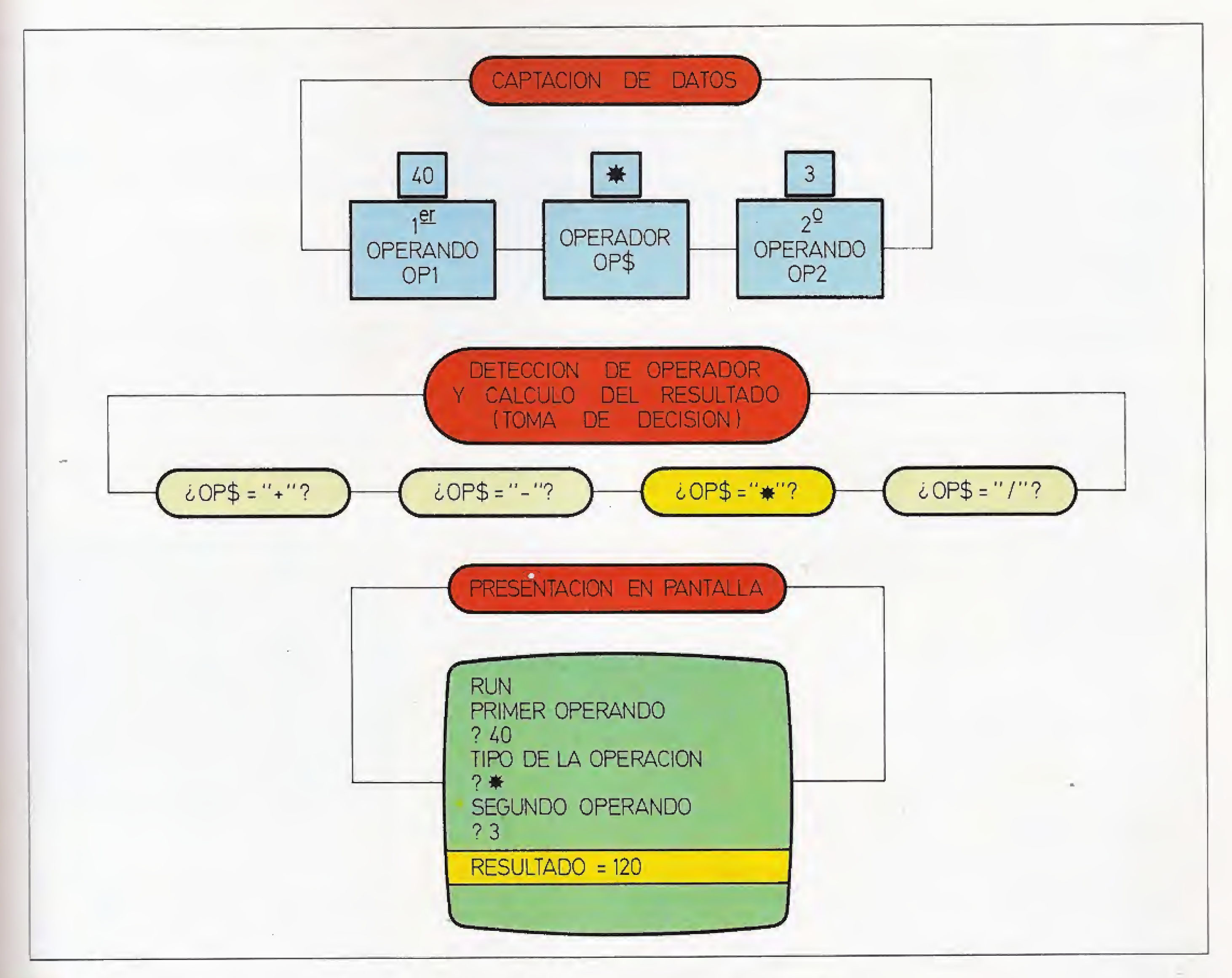
En definitiva: A<>B será *cierto* siempre que A=B sea *falso* y viceversa.

A su vez, los operadores mayor y menor también disponen de sus respectivos complementarios. Fijémonos en el primero: ¿Cuándo no será cierto A>B? Naturalmente, cuando A sea inferior a B. Pero ¡cuidado!, también cuando A y B sean iguales. El operador capaz de evaluar esta condición se denomina «menor o igual» y su signo característico es «<=».

3<=3 ...CIERTO
3<=4 ...CIERTO
5<=2 ...FALSO

El último operador de relación es el complementario de «menor que» (<). Este debe ser *cierto* cuando el resultado de «<» sea *falso.* Análogamente al caso anterior existen dos posibilidades: que el primer dato sea mayor que el segundo o que ambos sean iguales. Por lo tanto, este operador se identificará como «mayor o igual». Su correspondiente signo es «>=», como era de esperar.

3>=3 ...CIERTO
3>=4 ...FALSO
5>=2 ...CIERTO



La figura refleja el funcionamiento del programa BASIC adecuado para convertir al ordenador en una calculadora. Las tres zonas básicas del programa se concretan en la captación de datos, la detección del operador y el cálculo del resultado, y en la presentación de éste en pantalla.

Un poco de lógica

El resultado de evaluar la condición impuesta en una instrucción IF, es un dato de tipo lógico. Sabemos que el ordenador trabaja internamente con datos expresados a base de ceros y unos. Estos no son ni más ni menos que los mismos datos introducidos, aunque codificados de forma que sean inteligibles para la máquina. Por suerte, el programador no tiene por qué conocer esta compleja representación interna de los datos. Cabe, sin embargo, señalar un punto relacionado con el tema que nos ocupa: la evaluación de condiciones IF. El resultado de dicha evaluación —lo que hemos dado en llamar dato de tipo lógico—, admite dos valores, a saber: *cierto* o *falso*. Estos se codifican en el ordenador en forma de «1» si la respuesta es *cierto*, ó «0» en el caso contrario *(falso)*.

Desde luego, estos datos pueden ser almacenados en variables numéricas, lo que se consigue mediante una sentencia de asignación, en la cual la expresión a evaluar irá entre paréntesis. Por ejemplo:

```
POSITIVO=(A>=0)
```

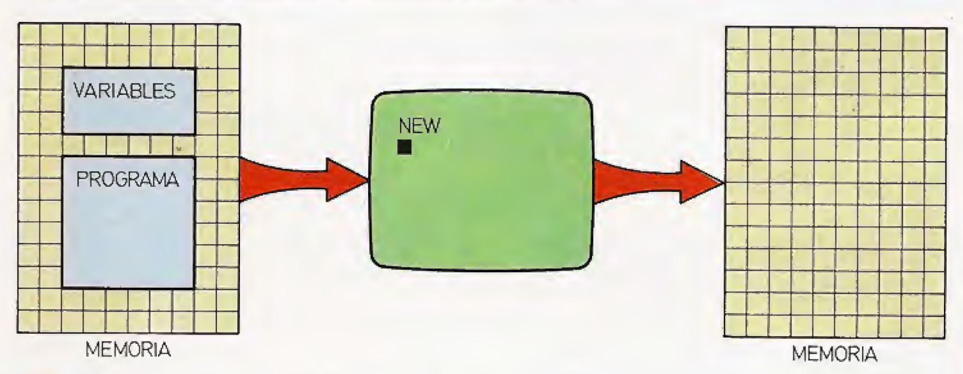
Si el contenido de la variable A es un número positivo, o incluso el valor 0, la variable POSITIVO adoptará el valor

NEW

Borra el programa que se encuentra en la memoria del ordenador, así como todas las variables definidas hasta el momento.

Formato: NEW

Ejemplo: NEW



NEW es un comando BASIC que se utiliza en forma de instrucción directa (sin número de línea). Su misión es borrar de la memoria del ordenador el programa BASIC y las variables almacenadas en ella.

cierto (1), en el caso contrario adoptará el valor *falso* (0).

Las variables que contienen datos de tipo lógico, resultantes de evaluar una expresión, pueden utilizarse como condiciones dentro de una instrucción IF. Por ejemplo:

```
10 IF POSITIVO THEN ...
```

En este caso, POSITIVO debe contener un dato de tipo lógico *(cierto* o *falso:* 1 ó 0 en su expresión binaria).

Empleando este método, se pueden simplificar las expresiones de la condición dentro de las instrucciones IF/THEN/ELSE. Así, por ejemplo, es posible introducir condiciones como la que sigue:

```
10 IF A THEN ...ELSE...
```

en donde A es una variable numérica. En este caso, cuando la variable A adopte el valor 0, por efecto de la ejecución del programa, se ejecutará la zona ELSE de la instrucción. Por el contrario, cuando tome el valor 1 u otro cualquiera distinto de 0, se ejecutarán las instrucciones localizadas en la zona THEN, al igual que si la variable A tuviera en ese instante el valor lógico *cierto*.

Así pues, es perfectamente factible utilizar una variable numérica como condición en la sentencia IF. Esta es una alternativa particularmente útil cuando se desea realizar una acción siempre que determinado dato *no* sea nulo. Por ejemplo, si se trata de evitar que el ordenador realice una división por cero, puede adoptarse la siguiente instrucción:

```
50 IF DATO THEN SOLU=100/DATO
```

El cociente (zona THEN) sólo se ejecutará cuando DATO sea cierto. Como quiera que se trata de una variable numérica, sólo se tomará como falsa cuando su valor sea cero. Por lo tanto, la división se realizará únicamente si DATO es distinto de cero... precisamente, lo que se pretendía.

No hay que perder de vista que la referida instrucción es totalmente análoga a:

```
50 IF DATO <> 0 THEN SOLU=100/DATO
```

El siguiente ejemplo pone en práctica los conceptos explicados. El programa calculará la raíz cuadrada del número introducido siempre y cuando éste sea positivo. Si se introduce un número negativo, el ordenador lo rechazará y solicitará la introducción de otro número (la raíz cuadrada de un número negativo es una operación errónea).

```
10 INPUT A
20 LET POSITIVO=(A>=0)
30 IF POSITIVO THEN GOTO 50
40 GOTO 10
50 LET B=SQR(A)
60 PRINT "LA RAIZ CUADRADA
   DE ";A; "ES: ";B
70 END
█
```

En la línea 20 se asigna un valor lógico a la variable POSITIVO. Esta variable se utiliza posteriormente como condición en la línea 30.

```
RUN
?4
LA RAIZ CUADRADA DE 4 ES: 2
RUN
?0
?25
LA RAIZ CUADRADA DE 25 ES: 5
█
```

Una calculadora a su servicio

A modo de resumen práctico de las instrucciones de salto condicional, va-

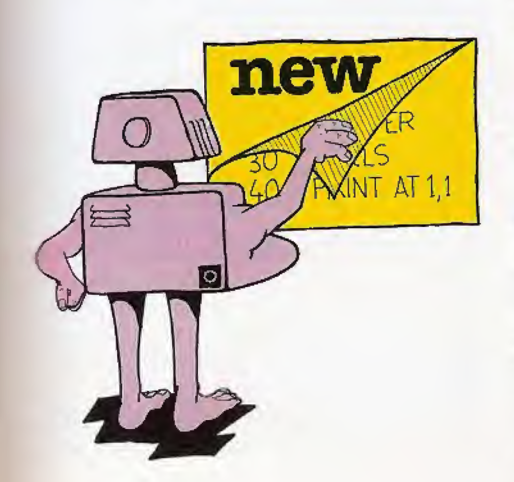
mos a construir un programa BASIC que convierta al ordenador en una calculadora; elemental pero adecuada para resolver las cuatro operaciones aritméticas. Su empleo se reducirá a introducir el primer operando, tras éste el tipo de operación y por último el segundo operando.

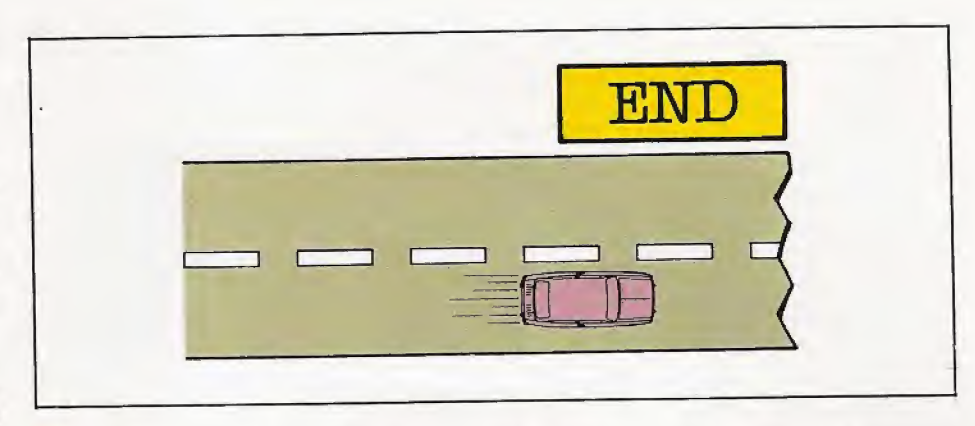
El primero de los operandos se capta a través de la instrucción INPUT OP1. La variable OP1 será la encargada de guardar el valor del primer operando.

El siguiente paso es introducir el tipo de operación que se desea realizar. Para no complicar el programa, las posibilidades se reducen a las cuatro operaciones básicas. Mediante la instrucción INPUT OP$, se lee el operador y se almacena en la variable: OP$. Por último, la instrucción INPUT OP2 se ocupará de leer el segundo operando que pasará a constituir el contenido de la variable OP2.

¿Qué hay que hacer para que el ordenador ejecute la operación solicitada? Muy fácil: sencillamente, poner en práctica las posibilidades de las instrucciones IF/THEN/ELSE. Estas deben ir comprobando las distintas operaciones y al coincidir el signo de operación con el contenido de OP$, ejecutar la referida operación con los datos OP1 y OP2. El resultado se guardará en una nueva variable cuyo nombre es RESUL.

Una vez realizado el cálculo, hay que presentar el resultado en la pantalla, por medio de una instrucción PRINT. Finalmente, la última línea del programa ordena un salto incondicional al principio del mismo; ello hará posible realizar sucesivos cálculos sin tener que ordenar una nueva ejecución.






La instrucción END se utiliza para poner fin a la ejecución de un programa. Indica a la máquina que la secuencia de instrucciones ha concluido y que, en consecuencia, puede abandonar el programa.

```
5 LET RESUL=0
10 PRINT "PRIMER OPERANDO"
20 INPUT OP1
30 PRINT "TIPO DE LA OPERACION"
40 INPUT OP$
50 PRINT "SEGUNDO OPERANDO"
60 INPUT OP2
70 IF OP$= "+" THEN LET RESUL=OP1+OP2
80 IF OP$="—" THEN LET RESUL=OP1—OP2
90 IF OP$= "*" THEN LET RESUL=OP1*OP2
100 IF OP$= "/" THEN LET RESUL=OP1/OP2
110 PRINT "RESULTADO="; RESUL
120 GOTO 10
```

Y si ya no es útil...

Sin duda alguna, el programa no será de mucha utilidad para quien posea una calculadora de bolsillo; por lo tanto, será necesario borrarlo de la memoria del ordenador, donde ha estado almacenado hasta ahora.

La solución drástica es, desde luego, desconectar la alimentación de la máquina. Este es un método nada recomendable; existen procedimientos más «elegantes» para borrar un programa. Uno de ellos es utilizar el comando NEW (nuevo). La ejecución de esta orden como instrucción directa, origina el borrado completo de todas las líneas del programa que hubiera en la memoria del ordenador, así como de todas las variables inicializadas y utilizadas hasta el momento. Además, el sistema queda inicializado a la espera de que un nuevo programa sea introducido a través del teclado, o por otro medio.

Interrupción de un programa

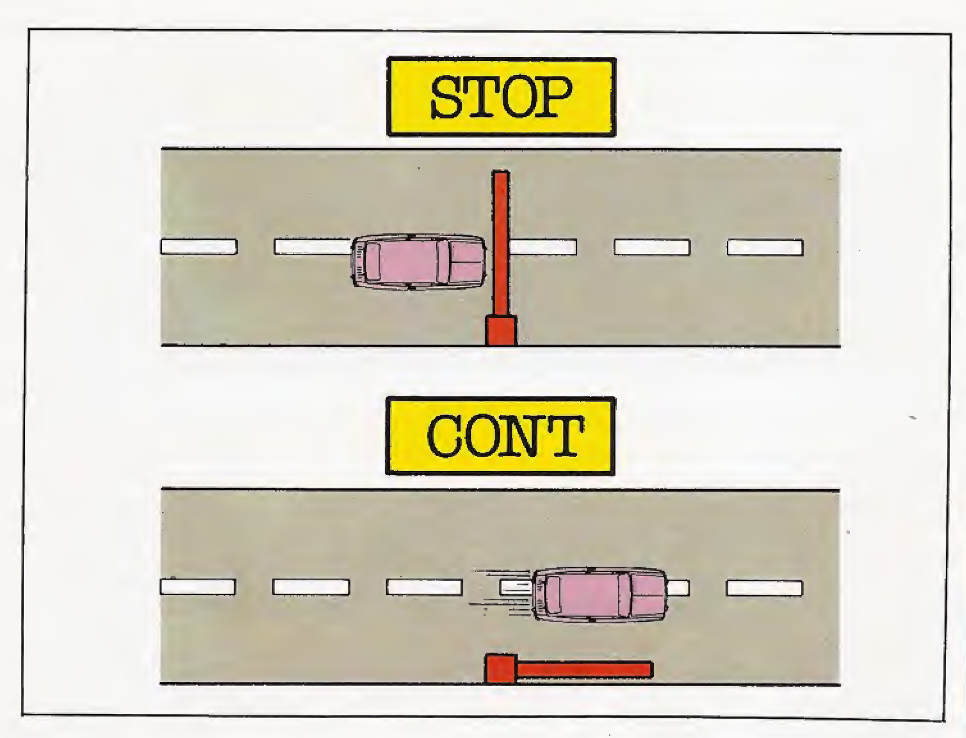
La ejecución de un programa evoluciona de forma secuencial y ordenada, comenzando por la línea de programa cuyo número es inferior. A partir de ésta se procede con la siguiente, respetando siempre el orden de menor a mayor en los números de las líneas ejecutadas.

Cuando el ordenador encuentra una instrucción END se detiene la ejecución. Esta instrucción suele colocarse en la última línea del programa. No obstante, nada impide situarla en cualquier otro punto del mismo. En tal caso, las líneas siguientes no se ejecutarán en la secuencia lógica, ya que el ordenador interpreta que el programa finaliza en la instrucción END.

El siguiente programa incluye dos líneas tras la instrucción END que no serán ejecutadas:

```
10 PRINT "LINEA 10"
20 PRINT "LINEA 20"
30 END
40 PRINT "LINEA 40"
50 PRINT "LINEA 50"

RUN
LINEA 10
LINEA 20
```



A diferencia con el comando END, STOP no supone la conclusión definitiva del programa, sino que sólo interrumpe la ejecución del mismo. Algo semejante a la colocación de una barrera en la trayectoria de un automóvil (secuencia de ejecución del programa). Por supuesto, una vez levantada la barrera por efecto del comando CONT, el automóvil puede continuar su avance (esto es, continuará la ejecución del programa).

El hecho de colocar una instrucción END en medio de un programa puede parecer absurdo, no obstante, llega a ser útil en ciertos casos; por ejemplo, si se desea interrumpir la ejecución bajo ciertas condiciones. El fragmento de programa que sigue dará por terminada la ejecución cuando la variable X tome el valor cero.

```
...
120 IF X <> 0 THEN GOTO 140
130 END
140 REM CONTINUACION
...
```

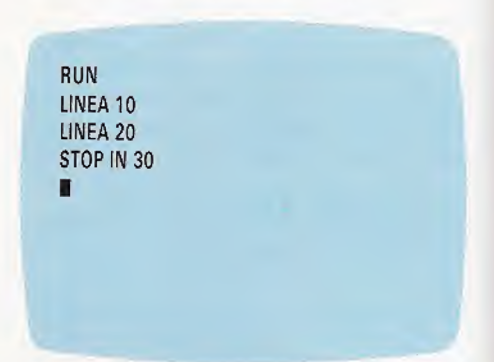
Las líneas de la 140 en adelante se ejecutarán exclusivamente cuando, al pasar por la comparación, X resulte distinto de 0. Si, por cualquier motivo, X fuera siempre nulo, las referidas líneas no se ejecutarían en ningún caso.

El comando STOP

En el BASIC existe otro comando destinado a interrumpir la ejecución; éste es el comando STOP. Al igual que END, puede situarse en cualquier punto del programa; e incluso puede reemplazar al propio END. Por ejemplo:

```
10 PRINT "LINEA 10"
20 PRINT "LINEA 20"
30 STOP
40 PRINT "LINEA 40"
50 PRINT "LINEA 50"
█
```

La ejecución del programa conducirá al siguiente resultado en la pantalla:



Al detenerse la ejecución, el comando STOP muestra un mensaje de parada (Stop, en inglés), en el que se indica la línea del programa en la que se ha producido la interrupción. Esta característica puede resultar útil a la hora de depurar un programa, colocando instrucciones STOP en puntos claves del mismo.

El ejemplo propuesto revela como única diferencia entre el uso de STOP o

STOP

Detiene la ejecución del programa en curso.

Formato: (Número de línea) STOP

Ejemplo: 50 STOP

END la indicación, en el primer caso, de la línea en la que se produce la interrupción. Sin embargo, STOP tiene otra peculiaridad que lo hace más potente. Esta característica está íntimamente relacionada con un nuevo comando: CONT.

El comando CONT

EL comando CONT se utiliza para reanudar la ejecución del programa interrumpido por medio de STOP.

CONT ha de utilizarse siempre en modo directo. Una vez introducido y tras accionar la tecla RETURN, la ejecución CONTinuará en la línea siguiente a aquella en la que se detuvo.

Veamos un ejemplo:

```
10 LET A=0
20 STOP
30 LET A=A+1
40 STOP
50 IF A=1 THEN LET A=2
60 STOP
70 GOTO 30
█
```

Con el programa anterior, el resultado sería el siguiente:

```
RUN

STOP IN 20
PRINT A
0
█
```

La ejecución se ha detenido en la línea 20, y se ha verificado el valor de la variable A. Ahora se reactiva la ejecución introduciendo la orden CONT:

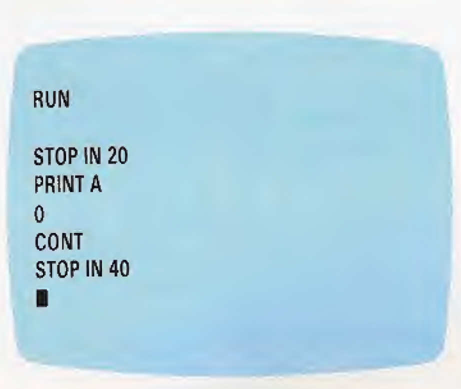
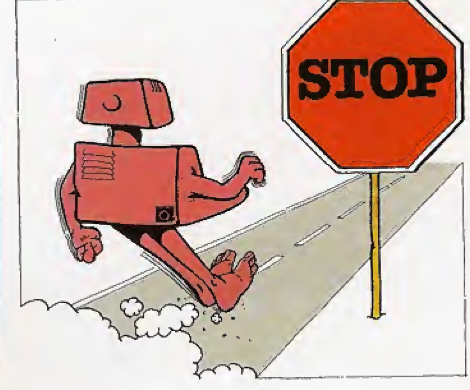


TABLA DE CONVERSION			
Ordenador	NEW	STOP	CONT
	NEW	STOP	CONT
AMSTRAD	NEW	STOP	CONT
APPLE II (APPLESOFT)	NEW	STOP	CONT
APRICOT (M-BASIC)	NEW	STOP	CONT
ATARI	NEW	STOP	CONT
CBM 64	NEW	STOP	CONT
DRAGON	NEW	STOP	CONT
EQUIPOS MSX	NEW	STOP	CONT
HP-150	NEW	STOP	CONT
IBM PC	NEW	STOP	CONT
MPF	NEW	STOP	CONT
NCR DM-V (MS-BASIC)	NEW	STOP	CONT
NEW BRAIN	NEW	STOP	CONT
ORIC	NEW	STOP	CONT
SHARP MZ-700 (MZ-BASIC)	NEW	STOP	CONT
SINCLAIR QL	NEW	STOP	CONTINUE
SPECTRAVIDEO	NEW	STOP	CONT
ZX-SPECTRUM	NEW	STOP	CONT

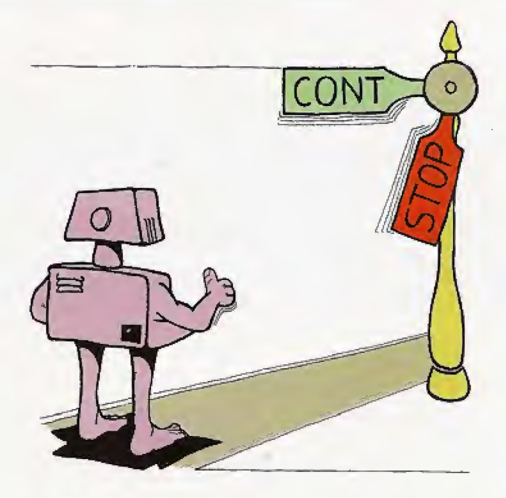


CONT

Reanuda la ejecución de un programa detenido por efecto del comando STOP.

Formato: CONT

Ejemplo: CONT



De nuevo puede visualizarse el contenido actual de A. Como es natural, el contenido actual de las variables no se ve alterado por el uso de STOP y CONT.

Con la ayuda de estos dos comandos es posible realizar un seguimiento del programa, interrumpiéndolo en el punto deseado y ordenando más tarde su reanudación.

Ambos comandos pueden también utilizarse para otros menesteres. Por ejemplo, a la hora de detener la ejecución del programa para apuntar datos intermedios. En ocasiones y dada la rapidez de cálculo del ordenador, los datos pueden aparecer en pantalla a muy alta velocidad; es entonces cuando se aconseja el uso de STOP. El siguiente ejemplo es un programa capaz de mostrar en pantalla los múltiplos de 11 a partir de 1000.

```
10 LET MUL=1001
20 PRINT MUL
30 LET MUL=MUL+11
40 GOTO 20
```

El bucle creado por medio de la instrucción GOTO 20 es muy rápido. En pocos instantes, la pantalla aparecerá repleta de múltiplos de 11; los primeros irán desapareciendo por la parte superior de la misma. Ello dificulta, cuando no hace imposible, su anotación.

Desde luego, existen otros metodos para «congelar» la presentación en pantalla, sin embargo, el más directo e inmediato es el que recurre al uso de STOP. Por ejemplo:

```
10 LET MUL=1001
20 PRINT MUL
25 STOP
30 LET MUL=MUL+11
40 GOTO 20
```

OPERADORES DE RELACION	
Operador	**Relación**
=	Igual
<	Menor que
>	Mayor que
<>	Distinto
<=	Menor o igual que
>=	Mayor o igual que

Ahora, la ejecución del programa se detendrá después de mostrar cada número sucesivo. Ello permitirá al usuario anotarlos, uno a uno, e introducir la orden CONT para pasar al siguiente dato.

El programa que sigue realiza la misma acción, si bien, su ejecución sólo se detendrá tras presentar cada grupo sucesivo de 20 números.

```
 10 REM MULTIPLOS DE 11
 20 LET MUL=1001
 30 LET C=0
 40 CLS
 50 PRINT "MULTIPLOS DE 11"
 60 PRINT
 70 PRINT MUL
 80 LET MUL=MUL+11
 90 LET C=C+1
100 IF C<20 THEN GOTO 70
110 STOP
120 GOTO 30
```

Las líneas 20, 70 y 80 actúan de forma similar a las del programa anterior. En este segundo programa la clave se encuentra en las líneas 90 y 100. La primera de ellas contabiliza en la variable C los números presentados. La línea 100, por su parte, evalúa la expresión C < 20, que será cierta mientras la cantidad de números visualizados sea inferior a 20. En dicha situación, el programa salta a la línea 70, imprimiendo el siguiente múltiplo.

La ejecución continúa con normalidad y de forma análoga hasta que se alcanza la cantidad de 20 datos en pantalla (C=20); instante en el que la condición deja de ser cierta y, en consecuencia, deja de producirse el salto a la línea 70. En su lugar, la ejecución prosigue en la línea inmediatamente posterior en la que se encuentra el comando STOP. Así pues, el programa se detiene únicamente tras presentar en pantalla un bloque de 20 datos.

El uso del comando CONT conduce a la reanudación del programa en la línea 120, de donde bifurcará hacia la 30. Esta última reinicia el contador C a cero, con lo que el programa queda en disposición de presentar un nuevo grupo de veinte múltiplos de once.

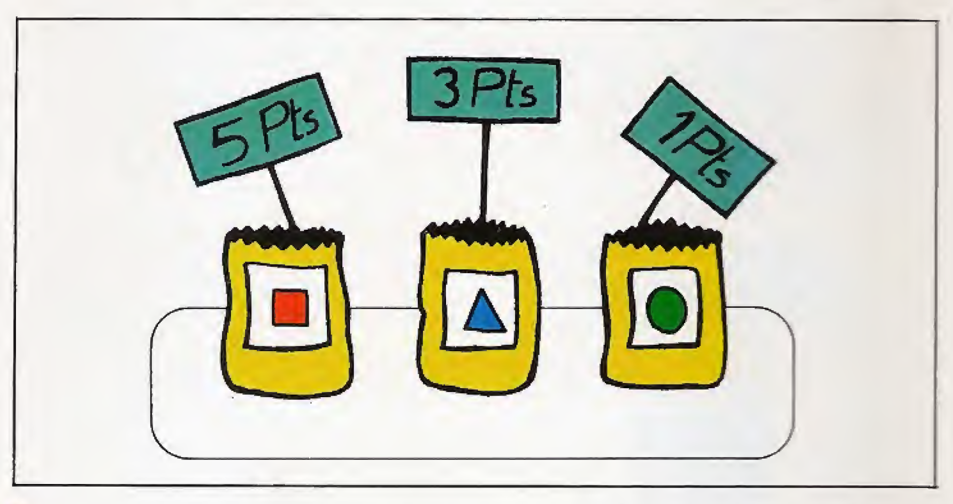
Programando bucles

Estructuras cíclicas y decisiones de alternativa múltiple



la hora de adentrarse en los secretos de la programación hay que disponerse a franquear con éxito el estudio de las estructuras de control. Este es uno de los aspectos más importantes de la programación en lenguaje BASIC.

Las estructuras de control llevan asociadas una serie de instrucciones cuya función primordial es la ruptura de la secuencia de ejecución normal del programa. Estas instrucciones permiten crear programas capaces de tomar decisiones en base a criterios establecidos por el programador. El resultado de la decisión puede llevar a ejecutar o no una determinada zona del programa, o bien puede controlar la ejecución repetitiva de una rutina un determinado número de veces.



El simple cálculo del importe total de una compra de tres artículos, de distinto precio unitario, constituye un ejemplo adecuado para estudiar las ventajas que aportan los bucles o estructuras cíclicas a la programación BASIC.

Programación de bucles

El objetivo de un programa BASIC puede ser, sencillamente, calcular el importe total de tres artículos de distinto precio unitario y adquiridos en distinta cantidad. La forma de escribir o codificar un programa semejante, empleando el lenguaje BASIC, no plantea excesivos problemas. Una posible solución es la siguiente:

```
 10 LET T=0
 20 INPUT "CANTIDAD DEL ARTICULO"; C
 30 INPUT "PRECIO DEL ARTICULO"; P
 40 LET N=C*P
 50 PRINT "CANTIDAD A PAGAR:"; N
 60 LET T=T+N
 70 INPUT "CANTIDAD DEL ARTICULO"; C
 80 INPUT "PRECIO DEL ARTICULO"; P
 90 LET N=C*P
100 PRINT "CANTIDAD A PAGAR:"; N
110 LET T=T+N
120 INPUT "CANTIDAD DEL ARTICULO"; C
130 INPUT "PRECIO DEL ARTICULO"; P
140 LET N=C*P
150 PRINT "CANTIDAD A PAGAR:"; N
160 LET T=T+N
170 PRINT "EL TOTAL A PAGAR ES:"; T; "PESETAS"
180 END
```

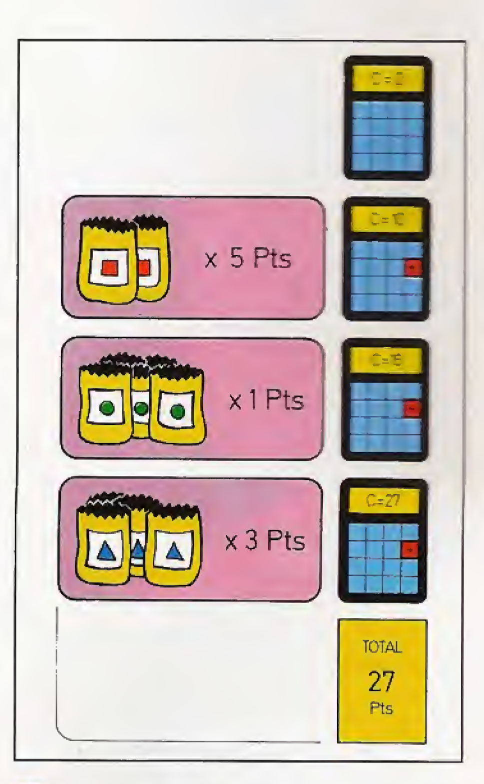
El programa resulta un tanto extenso aunque, sin lugar a dudas, realiza la función encomendada.

La primera instrucción (10 LET T=0) pone a cero la variable que irá totalizando los importes o cantidades a pagar por los diversos artículos adquiridos. La zona delimitada por las instrucciones 20 a la 60 solicita la introducción de la cantidad (C) y precio (P) del primero de los artículos; tras ello, calcula el importe total del mismo (40 LET N=C*P) y lo almacena en la variable T (60 LET T=T+N).

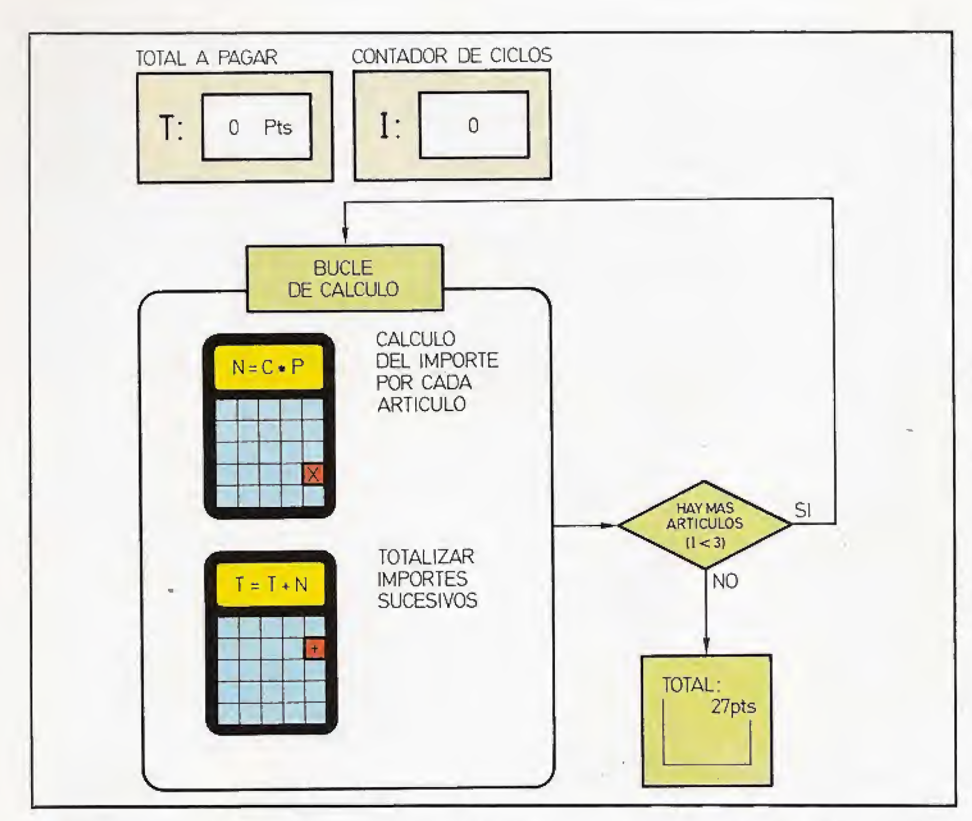
Como quiera que son tres los artículos que se adquieren, el programa repite la mencionada zona (instrucciones 20 a la 60) dos veces más (líneas 70 a la 110 y 120 a la 160). Por último ya sólo queda por presentar en la pantalla el importe total de la compra, memorizado en la variable T; de ello se encarga la instrucción 170.

Una solución más elegante, basada en el comando GOTO, sería:

```
10 LET T=0
20 LET I=0
30 INPUT "CANTIDAD DEL ARTICULO"; C
40 INPUT "PRECIO DEL ARTICULO"; P
50 LET N=C*P
60 PRINT "CANTIDAD A PAGAR"; N
70 T=T=N
```



El método más directo consiste en calcular por separado el importe total de cada artículo y totalizar la suma final. El programa al efecto no incorpora bucle alguno, sino que repite por tres veces la misma operación antes de evaluar el resultado final.



Con la ayuda de la instrucción IF/THEN es posible reducir la longitud del programa. El mismo bucle de cálculo se ejecutará en tres ocasiones, una para cada artículo. Al salir del ciclo, el programa sumará los importes parciales.

```
80 LET I=I+1
90 IF I<3 THEN GOTO 30
100 PRINT "LA CANTIDAD TOTAL A PAGAR ES:"; "PESE-
    TAS"
110 END
```

A todas luces, el nuevo programa mejora al anterior. En primer lugar ahorra tiempo al programador a la hora de confeccionarlo y, por otra parte, el programa resulta bastante más versátil. Con un mínimo esfuerzo es posible acondicionar el programa para que sea capaz de totalizar la adquisición de cualquier número de productos distintos. Para ello, sólo hay que definir cuántas veces debe repetirse el bucle de cálculo.

El número de veces que hay que repetir la rutina comprendida entre las líneas 30 y 70 es controlable actuando en distintos puntos del programa: modificando el valor inicial de I (línea 20), alternado su valor final (línea 90), o incluso variando el incremento que experimenta la variable I en cada nueva ejecución de la rutina (llínea 80).

Este ha sido un ejemplo ilustrativo de la ventaja que supone el empleo de instrucciones de control a la hora de realizar tareas repetitivas. No obstante, las posibilidades de esta técnica no se reducen a lo expuesto. El ejemplo siguiente revela con mayor elocuencia las ventajas que aporta este método, aplicándolo a un programa capaz de obtener una lista de los números pares, desde el 0 al 100:

```
10 LET C=0
20 PRINT I*2
30 I=I+1
40 IF I<50 THEN GOTO 20
50 END
█
```

En esta ocasión, la variable que se va incrementando, y que a partir de ahora llamaremos variable *contadora,* interviene directamente en los cálculos. Esta posibilidad ofrece ventajas muy interesantes en determinados casos.

La estructura FOR/NEXT

Se observa pues que las tareas rutinarias pueden ejecutarse cómodamente en el seno de *bucles.* Estos consisten, sencillamente, en un bloque de instrucciones que se ejecutan una y otra vez, hasta que se cumple una determinada condición.

La estructura más sencilla para la ejecución de bucles, y a su vez la más utilizada, es la FOR/NEXT. Esta queda definida en la figura adjunta, en la que aparece un diagrama de flujo representativo de su actuación.

La instrucción o instrucciones que componen la referida estructura FOR/NEXT deben contemplar cuatro elementos esenciales:

— Delimitación de los extremos del bucle; hay que precisar el principio y final del mismo.

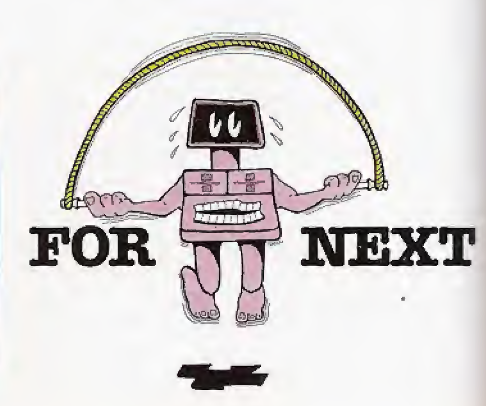
— Indicación de los valores inicial y final de la variable que se utilizará como contador.

— Definición del incremento o, de forma más general, de la variación que ha de experimentar la variable contadora con cada nueva ejecución del bucle.

— Indicación de la variable que va a utilizarse como contador.

En el BASIC, todo ello queda plasmado dentro de una estructura formada por dos instrucciones: FOR y NEXT.

En el argumento de la instrucción



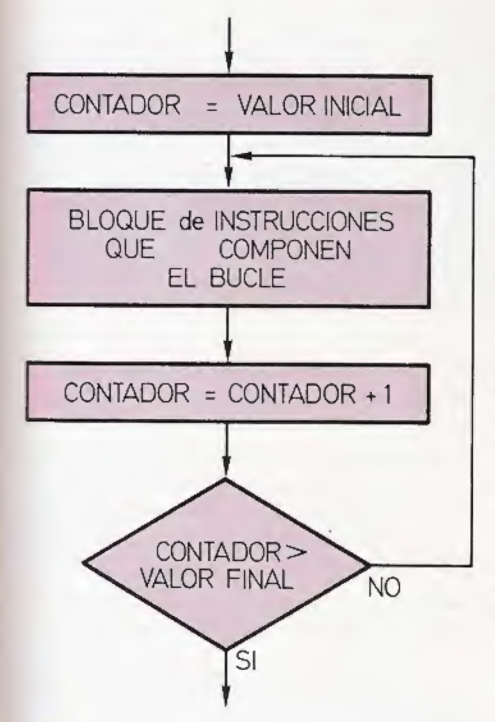


Diagrama de flujo representativo del funcionamiento de una estructura de bucle del tipo FOR/NEXT.

FOR hay que especificar cuál será la variable contadora, indicar el valor inicial y final de la misma (TO) y, además, precisar el incremento que experimentará dicha variable (STEP) en cada nueva ejecución del bucle. Hay que tener en cuenta que la variación que debe afectar a la variable contadora al completar cada ciclo puede ser incluso negativa.

El formato general de la instrucción FOR es el siguiente:

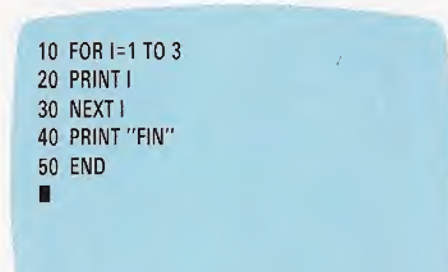
FOR <variable>=<valor inicial>TO <valor final> [STEP <incremento>]

De todos los elementos citados al principio, sólo queda por concretar el que se refiere a la delimitación del bucle. Ello se consigue fácilmente colocando una instrucción NEXT al final del bloque de instrucciones que constituyen el bucle. Su formato es tan simple como el que sigue:

NEXT <variable>

Por supuesto, la variable ha de coincidir con la utilizada como variable contadora en el argumento de la correspondiente instrucción FOR.

Veamos un sencillo ejemplo:



```
10 FOR I=1 TO 3
20 PRINT I
30 NEXT I
40 PRINT "FIN"
50 END
█
```

FOR/NEXT

Ejecuta el bucle encerrado entre ambas instrucciones tantas veces como sea necesario: hasta que la variable contadora alcance el valor de la expresión 2, partiendo el valor inicial establecido por la expresión 1 y en incrementos sucesivos dados por la expresión 3.

Formato: FOR <variable>=<expr. 1> TO <expr.2> STEP <expr.3>

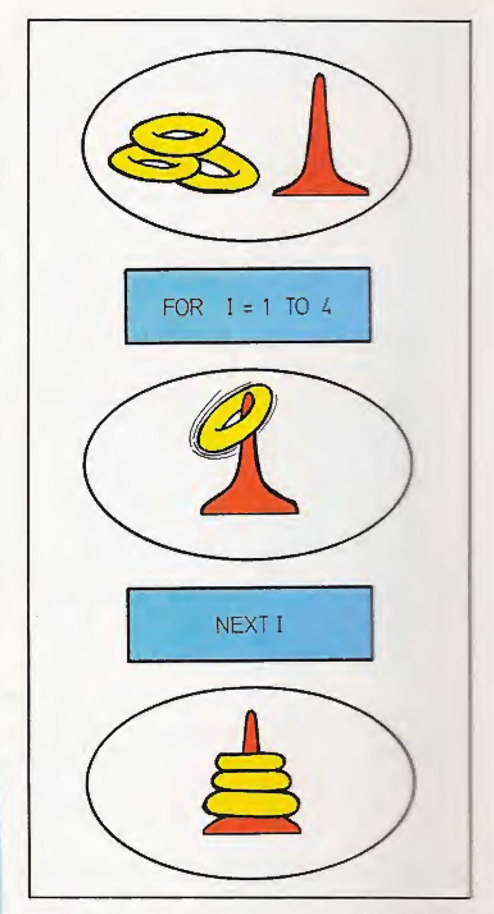
NEXT <variable>

Ejemplos: FOR I=1 TO 5 STEP 1.5

NEXT I

FOR K=A TO B STEP -Z

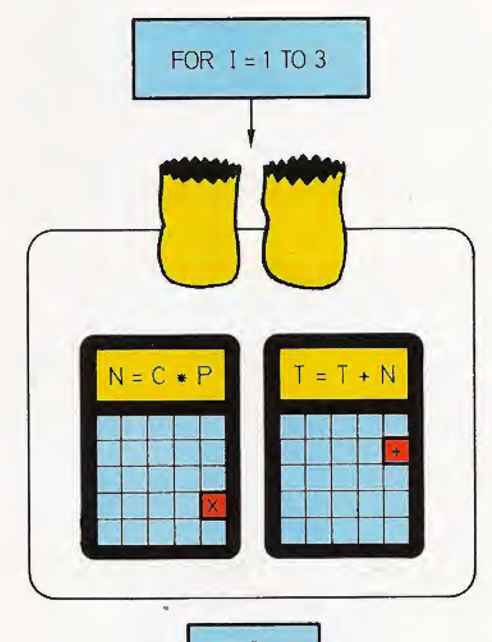
NEXT K



La estructura de control FOR/NEXT facilita la programación de tareas repetitivas. En la primera línea de la mencionada estructura (FOR/TO/STEP) se definen los valores inicial y final, además del incremento, de la variable que contabilizará los recorridos a través del bucle. Por su parte, la instrucción NEXT pondrá fin al conjunto de instrucciones que constituyen el bucle, devolviendo el control a la primera línea.

Al ejecutar el programa se observa que el bucle, constituido en este caso por una única instrucción (20 PRINT I), es ejecutado tres veces.

Tras recibir la orden RUN, se ejecuta por primera vez la instrucción 10. Esta dicta al ordenador que debe repetir la ejecución del bucle mientras que el valor de la variable contadora I (inicializada a 1) no exceda del límite 3. En este caso, al omitir la indicación del incremento sucesivo que debe afectar a I



Aplicando la instrucción FOR/NEXT al ejemplo inicial se reducirá la amplitud del programa adecuado para el cálculo del importe total de una compra. El mismo bucle de programa se ocupará de evaluar el importe de cada artículo a la cantidad adquirida en cada caso.

(zona STEP), la máquina interpretará que debe incrementarla en una unidad tras cada recorrido del bucle.



```
RUN
1
2
3
FIN
■
```

El valor del incremento puede ser cualquiera, incluso negativo, con lo que se convertiría en un decremento del valor de la variable contadora. En este último caso, la cifra colocada como valor inicial ha de ser mayor que la correspondiente al valor final.

A continuación aparece un ejemplo que establece un bucle controlado por la variable X. El bucle debe repetirse sucesivamente hasta que X adopte un valor superior a 1.5, teniendo en cuenta que tras completar cada ciclo, el valor de X se incrementará en 0.1 unidades. (Tal como es habitual en BASIC, la coma decimal se sustituye por el punto de acuerdo a la nomenclatura inglesa.)

La ejecución revela que el bucle se recorre en seis ocasiones, mostrando en cada caso el correspondiente valor de la variable X.

```
10 FOR X=1 TO 1.5 STEP 0.1
20 PRINT X;
30 NEXT X
40 END
RUN
1 1.1 1.2 1.3 1.4 1.5
■
```

Una de las opciones que otorga aún mayor flexibilidad a esta estructura, deriva de la posibilidad de sustituir alguno de los parámetros (valor inicial, valor final o incremento), o todos ellos, por una variable; el programa siguiente constituye un ejemplo elemental aunque ilustrativo de tal posibilidad:

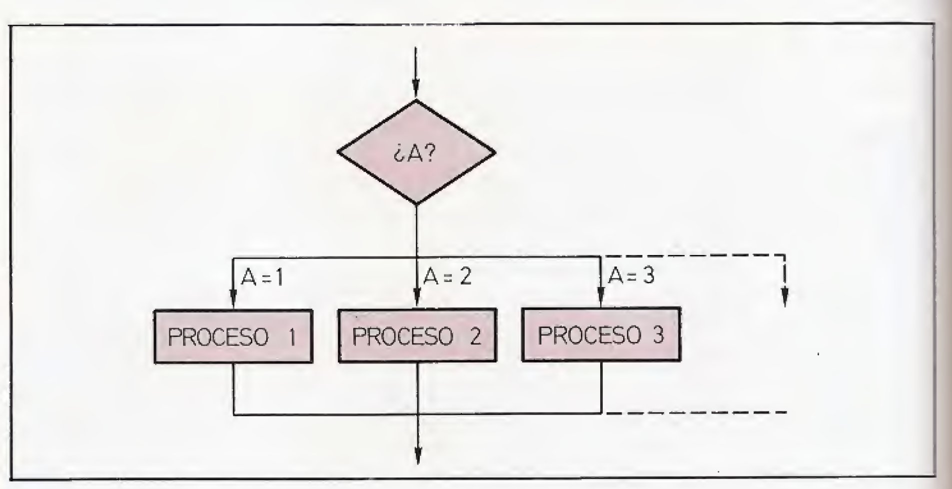
```
5 LET A=2
10 INPUT B
20 FOR I=A TO B
30 PRINT I
40 NEXT I
50 END
```

Tanto el valor inicial como el final lo aportan variables (A y B, respectivamente), cuyo valor puede asignarse en cualquier zona previa del programa.

La instrucción ON/GOTO

En ciertos casos, es necesario plantear decisiones con más de una alternativa. En la vida real, ello no plantea dificultades; sin embargo, en un ordenador la cosa no es tan fácil. Suponga, por ejemplo, que disponemos de un menú con varias opciones:

1. Imprimir las máximas puntuaciones.
2. Realizar una demostración.



El lenguaje BASIC incorpora una herramienta idónea para programar tomas de decisión de múltiple alternativa: la instrucción ON/GOTO.

ON/GOTO

Dependiendo del resultado de la expresión (1, 2, 3...) se poducirá un salto a la línea cuyo número aparezca en la posición correspondiente (primera, segunda, tercera...) dentro de la lista de números de línea que sigue a GOTO.

Formato: ON <expresión> GOTO <lista de números de línea>

Ejemplos: ON A GOTO 100, 200, 230, 400, 160

3. Empezar el juego.

Un menú de tres opciones que conduce a tres procesos distintos. Según el número de la opción elegida, el ordenador habrá de ejecutar una serie de operaciones, muy distintas en cada caso. Uno de los posibles métodos para programar esta toma de decisión en BASIC consiste, simplemente, en encadenar sucesivas instrucciones IF/THEN; por ejemplo:

```
10 INPUT A
20 IF A=1 THEN GOTO 500
30 IF A=2 THEN GOTO 1000
40 IF A=3 THEN GOTO 1500
50 PRINT "ESA OPCION NO ESTA EN EL MENU"
60 GOTO 10
█
```

El bloque de instrucciones empieza captando el número de la opción elegida a través de la instrucción INPUT A (línea 10). Acto seguido, las instrucciones 20, 30 40 detectan cada una de las tres posibles alternativas, y derivan la secuencia de ejecución hacia los subprogramas correspondientes a cada una de ellas. En efecto, se ha supuesto que los respectivos subprogramas se encuentran situados a partir de las líneas 500, 1000 y 1500.

El ejemplo termina con las instrucciones 50 y 60. La primera de ellas presentará un mensaje al usuario en el caso de que se introduzca un número que no corresponda a una de las opciones permitidas; tras ello, y por efecto de la instrucción GOTO 10, el ordenador aguardará a que se introduzca de nuevo el número de la opción.

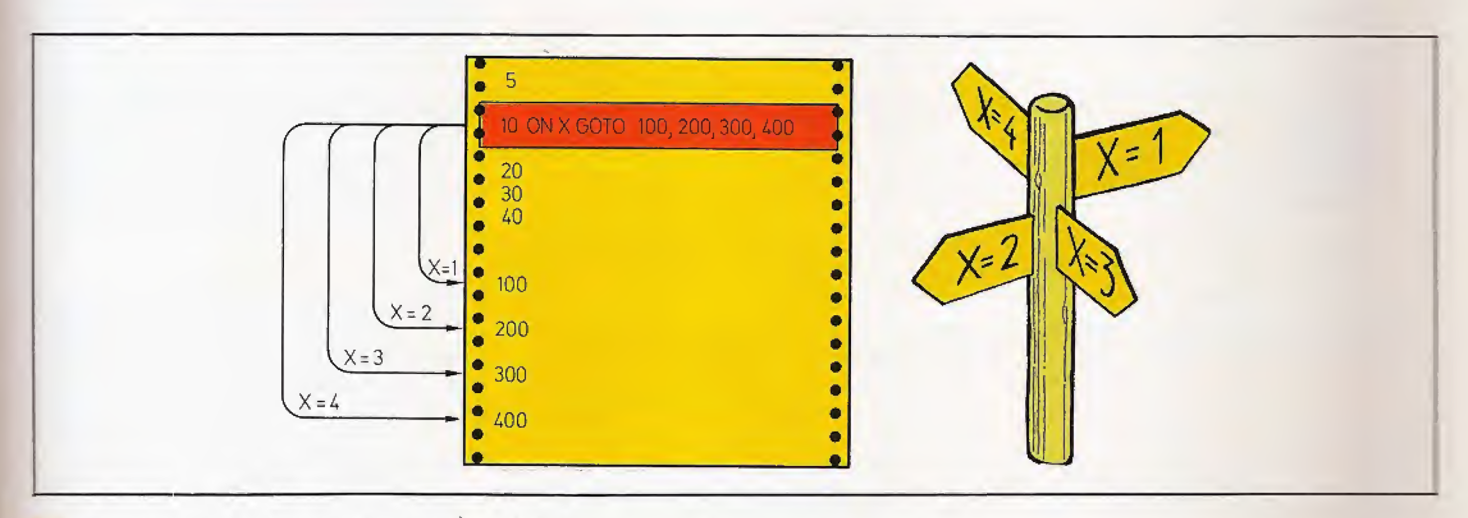
De la eficacia de esta solución no cabe la menor duda. No obstante, ¡qué sucederá si el número de opciones del menú se eleva a veinte o treinta? Hay que buscar otro procedimiento más cómodo y rápido.

Para programar este tipo de tomas de decisión de una forma más razonable, es necesario recurrir a estructuras lógicas más complejas. El lenguaje BASIC posee una instrucción especializada en tal cometido; ésta es ON/GOTO, cuyo formato general es el siguiente:



ON <expresión> GOTO <lista de números de línea>

Al utilizar la nueva instrucción, el programa anterior adoptará un nuevo aspecto:



Funcionamiento de la instrucción ON/GOTO. El valor que adopte la variable X o, en general, la expresión que sigue a ON, determinará el número de línea al que debe bifurcar el programa. Los números de línea que pueden verse afectados son los que aparecen en la lista incluida en la zona GOTO.

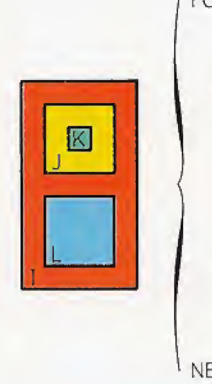


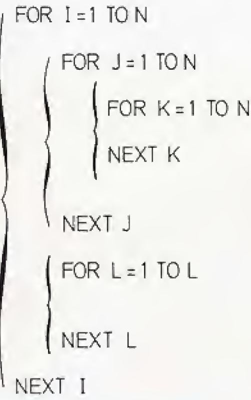
El teclado es la principal vía para la entrada de datos para el ordenador. Para gestionar su introducción, el BASIC cuenta con instrucciones especializadas; por ejemplo: las que pueden construirse a partir del comando INPUT.

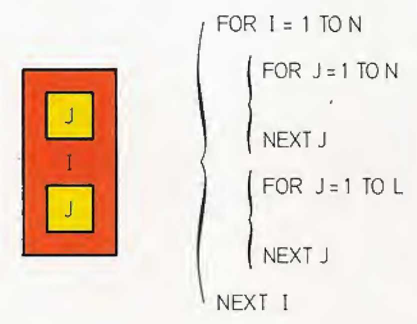
Bucles anidados

Los bucles anidados constituyen una técnica de programación que consiste en introducir bucles FOR/NEXT dentro de otros bucles de la misma naturaleza. Este es un método muy frecuentemente utilizado en las tareas de programación. Su puesta en práctica exige adoptar una precaución básica: los bucles deben "anidarse" de tal forma que cada uno de ellos esté completamente incluido dentro del otro. A su vez, dos bucles, uno anidado dentro del otro, no pueden compartir la misma variable contadora, sino que ésta ha de ser distinta. El programa adjunto ilustra la actuación de un bucle anidado.

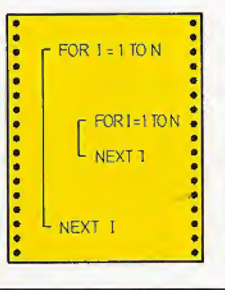
Las instrucciones 20, 30 y 40 constituyen un bucle FOR/NEXT anidado dentro de otro bucle exterior, este último delimitado por las líneas 10 y 50. El bucle externo utiliza la variable I como contador, mientras que el bucle anidado o interior emplea al respecto la variable J.







FOR I=1 TO N
FOR J=1 TO N
NEXT I
FOR I=1 TO N
NEXT J
NEXT I



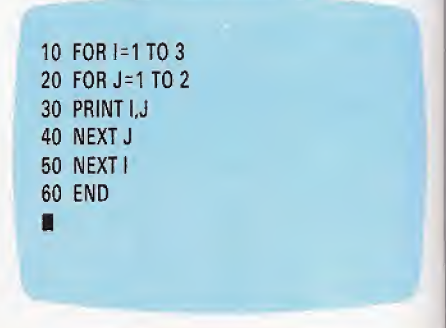
La instrucción 30 (PRINT I, J) permite observar cómo discurre la ejecución del programa. Para cada valor de la variable contadora I (bucle exterior), se ejecutan todos los posibles ciclos del bucle anidado (para J=1 y para J=2). Ello se debe a que el retorno del bucle externo (NEXT I) sólo se produce una vez que se ha salido del bucle interior y la ejecución rebasa la instrucción 40.

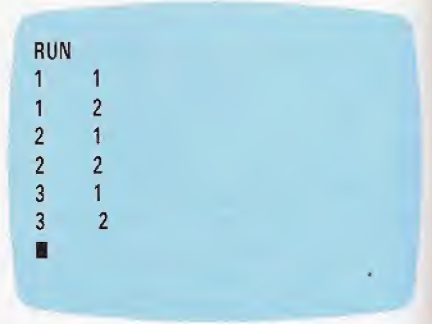
Algunos dialectos BASIC permiten cerrar los bucles anidados por medio de una sola instrucción NEXT, conteniendo ésta a todas las variables contadoras, ordenadas y separadas por comas.

En el ejemplo anterior, la instrucción de la línea 40 quedaría, en tal caso, de la siguiente forma:

40 NEXT J,I

Hay que tener en cuenta, no obstante, que esta posibilidad no es compartida por todos los dialectos BASIC; de ahí que, en principio, sea oportuno recurrir al método más general de cerrar cada bucle con una instrucción NEXT específica.





```
10 INPUT A
20 ON A GOTO 500, 1000, 1500
30 PRINT "ESA OPCION NO ESTA EN EL MENU"
40 GOTO 10
■
```

Sin lugar a dudas, el empleo de la instrucción ON/GOTO reduce la amplitud del programa; especialmente si pensamos en un caso en el que la decisión esté sujeta a un elevado número de opciones.

Las dos zonas que conforman a la nueva instrucción están perfectamente delimitadas. La primera de ellas la ocupa el comando ON y su argumento. Este contiene la expresión a evaluar, o sencillamente la variable cuyo valor condicionará la bifurcación o salto.

El comando GOTO y su argumento, integrado por una serie de números de línea separados por comas, dan cuerpo a la segunda zona de la instrucción. La relación entre ambas es muy simple. Tras evaluar el resultado de la expresión que acompaña a ON, el ordenador ejecutará un salto al número de línea que sigue a GOTO, cuya posición coincida con el valor extraido en la zona ON. Esto es: si el valor de la expresión es 1, el salto se producirá al primer número de línea; si es 2 al segundo; si su valor es 3 al tercero y así sucesivamente.

En el programa ejemplo, cuando se elija la opción 1 —tal será el valor de la variable A que ocupa la zona ON—, el salto conducirá a la línea 500; si se elije la opción 2, el salto llevará a la línea 1000; mientras que si la opción seleccionada es la 3, el programa bifurcará la línea 1500.

En definitiva, la elección se realiza a partir del valor de la expresión localizada en la zona ON. Si fuera necesario, el ordenador redondeará el valor del resultado al número entero más próximo, para que de esta forma quede precisado el orden del número de línea al que debe producirse el salto.

TABLA DE CONVERSION

Ordenador	FOR/NEXT		ON/GOTO
	FOR<variable>=<i> TO <f> STEP <r>	NEXT <variable>	ON <expresión> GOTO <lista>
AMSTRAD	FOR <variable>=<i> TO <f> STEP <r>	NEXT <variable>(1)	ON <expresión> GOTO <lista>
APPLE II (APPLESOFT)	FOR <variable><i> TO <f> STEP <r>	NEXT <variable>	ON <expresión> GOTO <lista>
APRICOT (M-BASIC)	FOR <variable>=<i> TO <f> STEP <r>	NEXT <variable>	ON <expresión> GOTO <lista>
ATARI	FOR <variable>=<i> TO <f> STEP <r>	NEXT <variable>	ON <expresión> GOTO <lista>
CBM 64	FOR <variable>=<i> TO <f> STEP <r>	NEXT <variable>	ON <expresión> GOTO <lista>
DRAGON	FOR <variable>=<i> TO <f> STEP <r>	NEXT <variable>	ON <expresión> GOTO <lista>
EQUIPOS MSX	FOR <variable>= <i> TO <f> STEP <r>	NEXT <variable>	ON <expresión> GOTO <lista>
HP-150	FOR <variable>= <i> TO <f> STEP <r>	NEXT <variable>	ON <expresión> GOTO <lista>
IBM PC	FOR <variable>= <i> TO <f> STEP <r>	NEXT <variable>	ON <expresión> GOTO <lista>
MPF	FOR <variable>= <i> TO <f> STEP <r>	NEXT <variable>	ON <expresión> GOTO <lista>
NCR DM-V (MS-BASIC)	FOR <variable>= <i> TO <f> STEP <r>	NEXT <variable>	ON <expresión> GOTO <lista>
NEW BRAIN	FOR <variable>= <i> TO <f> STEP <r>	NEXT <variable>	ON <expresión> GOTO <lista>
ORIC	FOR <variable>= <i> TO <f> STEP <r>	NEXT <variable>	ON <expresión> GOTO <lista>
SHARP MZ-700 (MZ-BASIC)	FOR <variable>= <i> TO <f> STEP <r>	NEXT <variable>	ON <expresión> GOTO <lista>
SINCLAIR QL	FOR <variable>= <i> TO <f> STEP <r>	NEXT <variable>	ON <expresión> GOTO <lista>
ESPECTRAVIDEO	FOR <variable>= <i> TO <f> STEP <r>	NEXT <variable>	ON <expresión> GOTO <lista>
ZX-SPECTRUM	**FOR** <variable>=<i> TO <f> STEP <r>	NEXT <variable>	ON <expresión> GOTO <lista>

<variable>: variable numérica. <i>:expresión que aporta el valor inicial. <f>: expresión que define el valor final. <r>: expresión que define el incremento. <lista>: lista de números de línea.

FORMULACIONES DE LOS COMANDOS

FOR <variable>: <i> TO <f> STEP <r>: define un bucle que se ejecutará repetidamente hasta que la variable alcance el valor de f, en sucesivos incrementos de valor r. NEXT <variable>: cierra el bucle definido por medio de una instrucción FOR. ON <expresión> GOTO <lista>: transfiere la secuencia de ejecución a uno de los números de línea que figuran en la lista, en función del valor de la expresión.

OBSERVACIONES:

(1) Es posible suprimir el nombre de la variable.

También puede ocurrir que el valor de la expresión sea cero, o mayor que el número de elementos en la lista; en tales circunstancias no se producirá salto alguno, sino que se ejecutará la instrucción siguiente a ON/GOTO.

En otro orden, puede suceder que el valor de la expresión sea negativo superior a 255; ambas situaciones harán que el ordenador presente un mensaje de error en la pantalla.

Simulación de la estructura ON/GOTO

La disparidad de los dialectos del lenguaje BASIC que equipan los distintos ordenadores se traduce en muy variados inconvenientes. Uno de ellos es la posible ausencia de la instrucción ON/GOTO, lo que, en principio, parece obligar al programador a olvidarse de esta eficaz herramienta para la toma de decisiones de múltiple alternativa. Aun cuando éste sea el caso, no hay por qué renunciar a esta útil estructura de control, ya que es posible simularla por medio de otras instrucciones BASIC más habituales. La descripción va a partir del siguiente ejemplo:

```
100 X=(RND*3)+1
110 ON X GOTO 200, 300, 400
200 PRINT "PRONOSTICO: 1"
210 GOTO 600
300 PRINT "PRONOSTICO: X"
310 GOTO 600
400 PRINT "PRONOSTICO: 2"
600 "¡SUERTE Y A POR LOS CATORCE!"
700 END
```

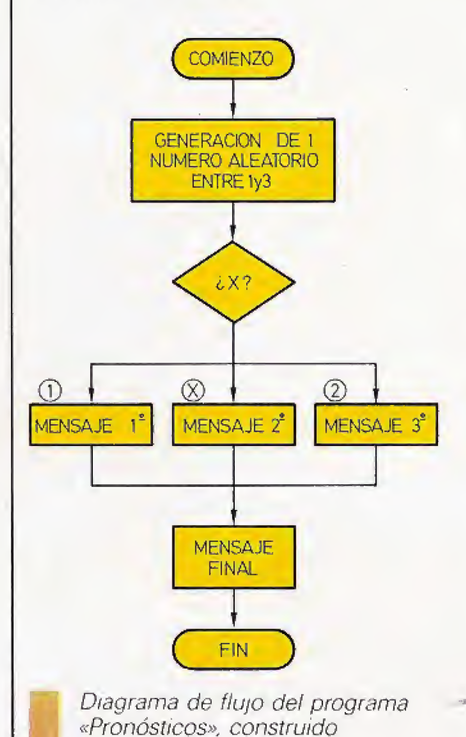


Diagrama de flujo del programa «Pronósticos», construido utilizando la estructura ON/GOTO.

Se trata de una zona de programa BASIC que incluye una toma de decisión múltiple, y cuyo correspondiente diagrama de flujo aparece en la figura adjunta. El cometido de este simple programa es la emisión de pronósticos para rellenar quinielas al azar.
La instrucción 100 incluye la función RND: una herramienta del vocabulario BASIC adecuada para la generación de números aleatorios y cuyo estudio se acometerá en un próximo capítulo de la obra. Por el momento, basta con saber que al ejecutar la referida instrucción la variable X adoptará un valor comprendido entre 1 y 3. Este valor sera el que utilizará la instrucción ON/GOTO para bifurcar hacia las líneas 200, 300 ó 400 y presentar los pronósticos 1, X ó 2, respectivamente.
La escritura de esta rutina omitiendo la instrucción ON/GOTO no plantea excesivos problemas. El método más inmediato es recurrir al empleo de sucesivas decisiones simples del tipo IF/THEN. Desde luego, el programa será más extenso, pero mantendrá su eficacia. Aquí esta una posible solución que consiste, sencillamente, en sustituir la línea 110 del programa inicial por las tres siguientes:

```
110 IF X=1 THEN GOTO 200
120 IF X=2 THEN GOTO 300
130 IF X=3 THEN GOTO 400
```

Al observar el funcionamiento de la primera instrucción del programa, salta a la vista que el valor de X siempre va a ser 1, 2 ó 3. De ahí que sea suficiente con detectar dos de los posibles valores, por ejemplo 2 y 3. Naturalmente, de no adoptar el valor 2 ó 3, X será igual a 1; en consecuencia, puede eliminarse la comparación con 1, de tal forma que su ejecución ocurra en el caso de no resultar positivas las otras dos detecciones.
Una vez eliminada la comparación de X con 1 —asumiendo que X no va a exceder en ningún caso del margen de valores esperados—, el programa quedará como sigue:

```
100 X=(RND*3)+1
110 IF X=2 THEN GOTO 300
120 IF X=3 THEN GOTO 400
200 PRINT "PRONOSTICO: 1"
210 GOTO 600
300 PRINT "PRONOSTICO: X"
310 GOTO 600
400 PRINT "PRONOSTICO: 2"
600 PRINT "¡SUERTE Y A POR LOS CATORCE!"
700 END
```

Su desarrollo es ilustrado por el correspondiente diagrama de flujo. Algunos ordenadores admiten una segunda alternativa. Para ello, han de cumplirse dos condiciones. En primer lugar, su traductor BASIC ha de permitir el empleo de variables y expresiones en el argumento de la instrucción GOTO; por otra parte, debe ser posible encontrar una relación matemática entre las diferentes líneas a las que se desea bifurcar, en función de los valores que tome la variable o expresión a evaluar.
En el ejemplo propuesto, la línea 110 —ocupada por la instrucción ON/GOTO— podría reemplazarse por la siguiente:

```
110 GOTO (X*100)+100
```

Tal como se observa, los posibles valores de X (1, 2 ó 3), provocarán el salto a las líneas 200, 300 y 400, respectivamente. Esta es una solución bastante más cómoda; sin embargo, hay que tener en cuenta que no siempre es posible aplicarla.

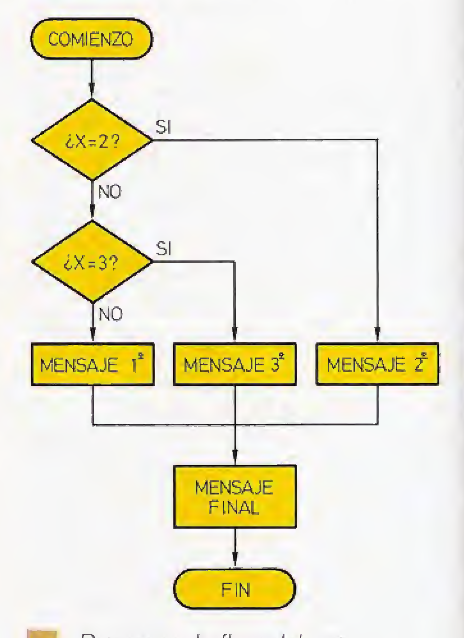


Diagrama de flujo del mismo programa, una vez sustituida la instrucción ON/GOTO por dos tomas de decisión del tipo IF THEN.

El ordenador como herramienta

La solución a tareas repetitivas



Una duda inmediata y compartida por los potenciales usuarios de un ordenador, es la que toma cuerpo en la siguiente pregunta: ¿de qué forma puede colaborar el ordenador en mi trabajo? La respuesta es sumamente compleja por lo extensa que puede resultar. Son tantas y tan variadas las tareas en las que tiene cabida el ordenador, que éste se convierte poco menos que en una máquina universal. Sin embargo, no hay que olvidar que, en definitiva, el ordenador no es más que una herramienta; un simple convidado de piedra si no cuenta con la instrucción del usuario.

Las tareas para las que está más capacitado un ordenador son aquellas que implican un gran volumen de operaciones repetitivas. Esta es una afirmación cuya elocuencia queda plasmada en el siguiente ejemplo.

El problema se localiza en una compañía de alquiler de vehículos. Las tarifas se establecen con arreglo a dos posibles fórmulas:

a) 8.000 pesetas diarias sin límite de kilómetros.

b) 3.000 pesetas diarias más un suplemento de 15 pesetas por kilómetro recorrido.

La eficaz herramienta que es el ordenador, debe ser capaz de construir una tabla comparativa del coste de cada opción para distintos kilometrajes.

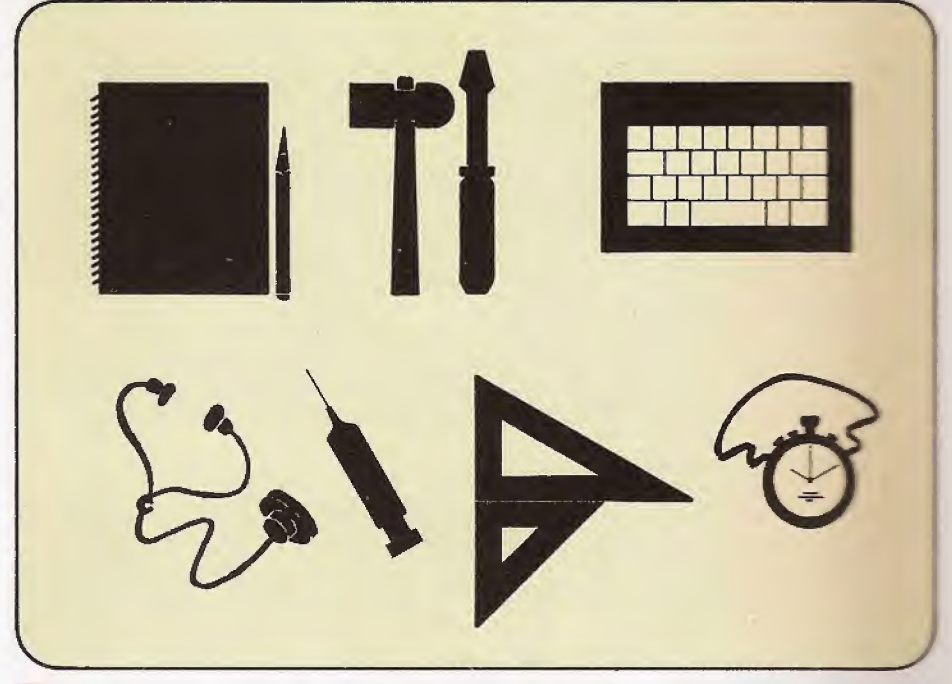
Denominando K a la variable que ha de contener el número de kilómetros recorridos, la cantidad a pagar de acuerdo a la segunda fórmula es:

```
P2=3000+K*15
```

Acogiéndose a la primera fórmula, la cantidad es siempre 8.000 pesetas, con independencia de los kilómetros recorridos.

En estas condiciones, se decide analizar los distintos costes variando el número de kilómetros. Tal variación comprenderá desde 50 a 500 kilómetros, con incrementos sucesivos de 25 kilómetros. Este es un cálculo repetitivo perfectamente encuadrable en un bucle del tipo FOR/NEXT:

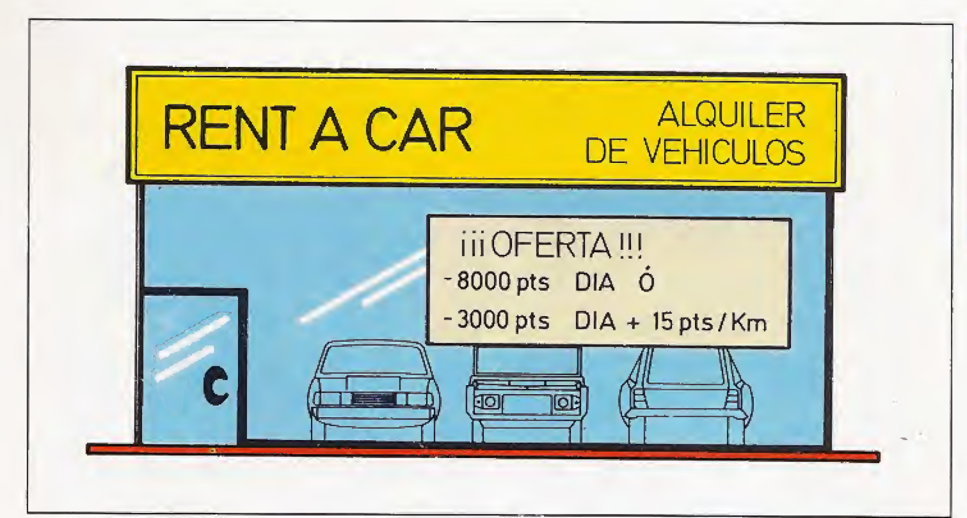
```
FOR K=50 TO 500 STEP 25
```



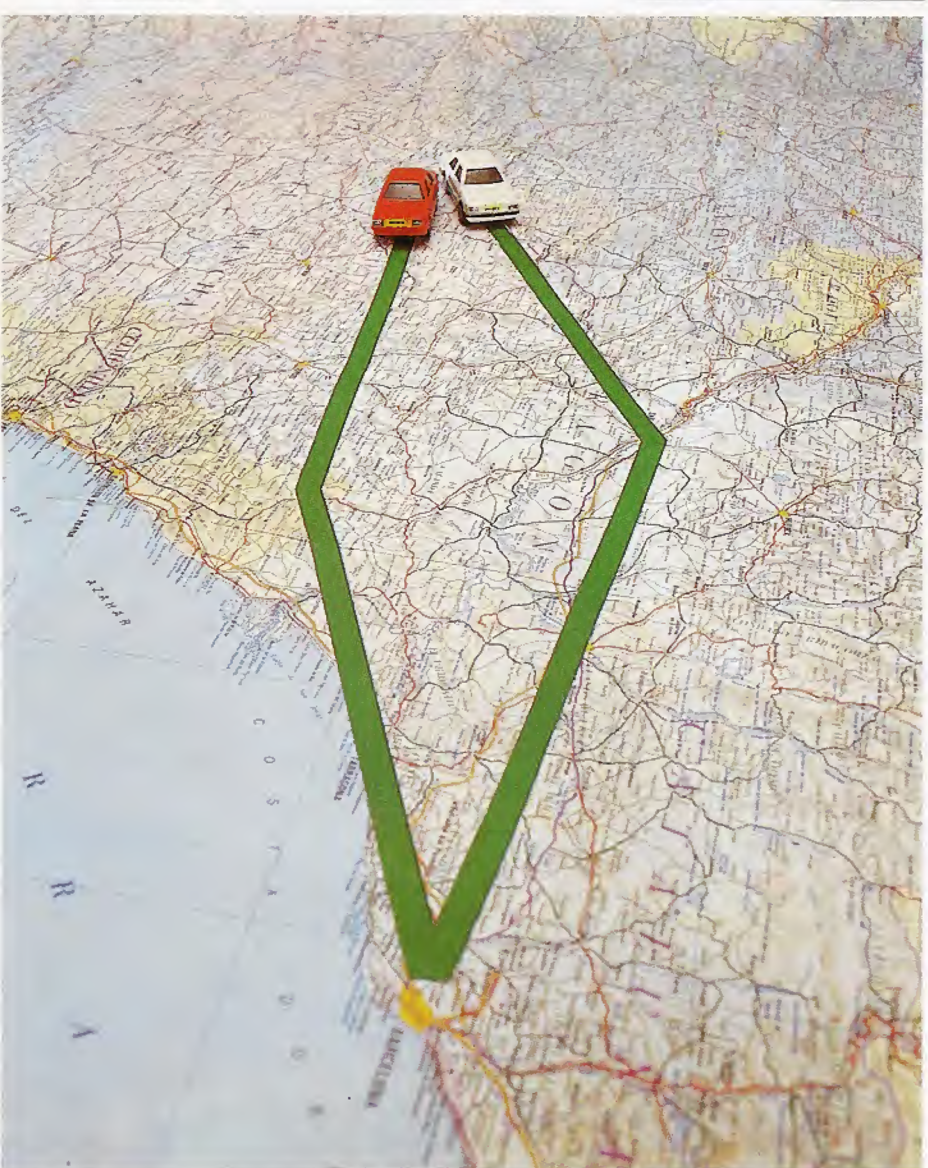
El ordenador es una eficaz herramienta al servicio del usuario. Una herramienta capaz de colaborar en las más diversas tareas; por supuesto siempre y cuando reciba la programación adecuada.



¿De qué forma puede colaborar el ordenador en mi trabajo? La amplitud de la respuesta tiene su límite en la capacidad del programador para «instruir» a la máquina.



El ordenador puede demostrar ampliamente su capacidad en actividades que exijan un gran volumen de operaciones repetitivas. Un ejemplo práctico lo aporta el ejemplo desarrollado en el texto. En él se realiza el cálculo comparativo del coste asociado a dos distintas modalidades de alquiler.



En la instrucción de apertura, 50 es el valor inicial, 500 el valor final y 25 el incremento para cada ejecución del bucle.

Una vez agrupadas las instrucciones oportunas, se llega al siguiente programa:

```
 1 REM ESTUDIO DE FORMULA DE ALQUILER
10 PRINT "KILOMETROS", "FORMULA1", "FORMULA2"
20 FOR K=50 TO 500 STEP 25
30 PRINT K, 8000, 3000+15*K
40 NEXT K
50 END
```

La línea 30 escribirá en primer lugar el valor de la variable K, seguido por el valor constante 8000 y, por último, el resultado de calcular 3000+15*K.

Si debido a la especial versión de su traductor BASIC, el ordenador no saltara los espacios adecuados al uso de las comas en la instrucción PRINT, puede sustituirse la línea 30 por la siguiente:

```
30 PRINT K;"";8000;"";3000+15*K
```

Y la 10 por:

```
10 PRINT "KM. -FOR.1 -FOR.2"
```

Tras ejecutar el programa, aparecerá en la pantalla el siguiente resultado.

El ordenador debe ser capaz de construir una tabla comparativa del coste de cada fórmula de alquiler para distintos kilometrajes.

```
RUN <CR>
KILOMETROS   FORMULA1   FORMULA2
50           8000       3750
75           8000       4125
100          8000       4500
125          8000       4875
150          8000       5250
175          8000       5625
200          8000       6000
225          8000       6375
250          8000       6750
275          8000       7125
300          8000       7500
325          8000       7875
350          8000       8250
375          8000       8625
400          8000       9000
425          8000       9375
450          8000       9750
475          8000       10125
500          8000       10500
```

A la vista de la tabla se puede calcular el número de kilómetros más rentable para cada una de ambas fórmulas de alquiler. Sin embargo, ésta es una tarea que muy bien puede realizar el propio ordenador.

Se observa que recorriendo poco kilómetros la segunda fórmula resulta más económica. Así pues, lo que interesa es conocer a partir de qué kilometraje esta fómula se hace más cara que la primera.

Un primer bloque de instrucciones capaz de evaluar tal circunstancia es el que aparece a continuación:

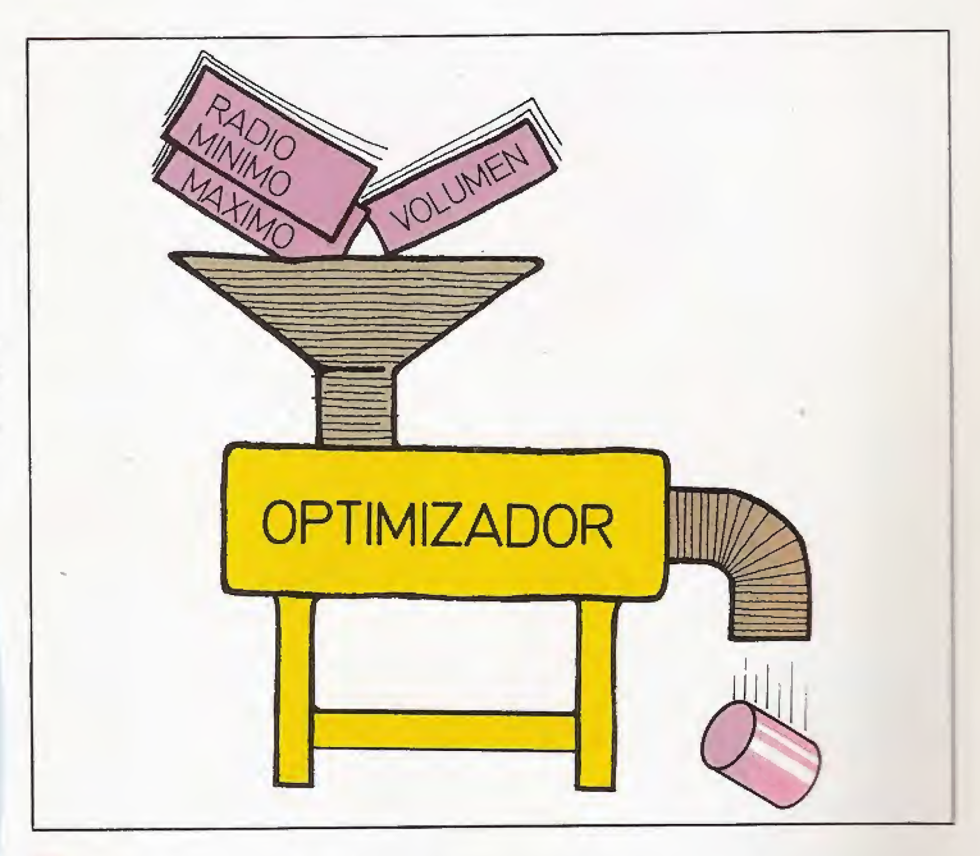
```
10 K=50
20 P2=3000+15*K
30 IF P2>8000 THEN PRINT K
40 K=K+25
50 GOTO 20
█
```

En el mismo, se van incrementando los kilómetros en la línea 40, de 25 en 25 unidades. El número de kilómetros inicial se fija en 50 dentro de la línea 10. Tal como está planteado el programa, éste escribirá todos los valores K para los que el importe según la fórmula 2 supere el precio que establece la opción 1.

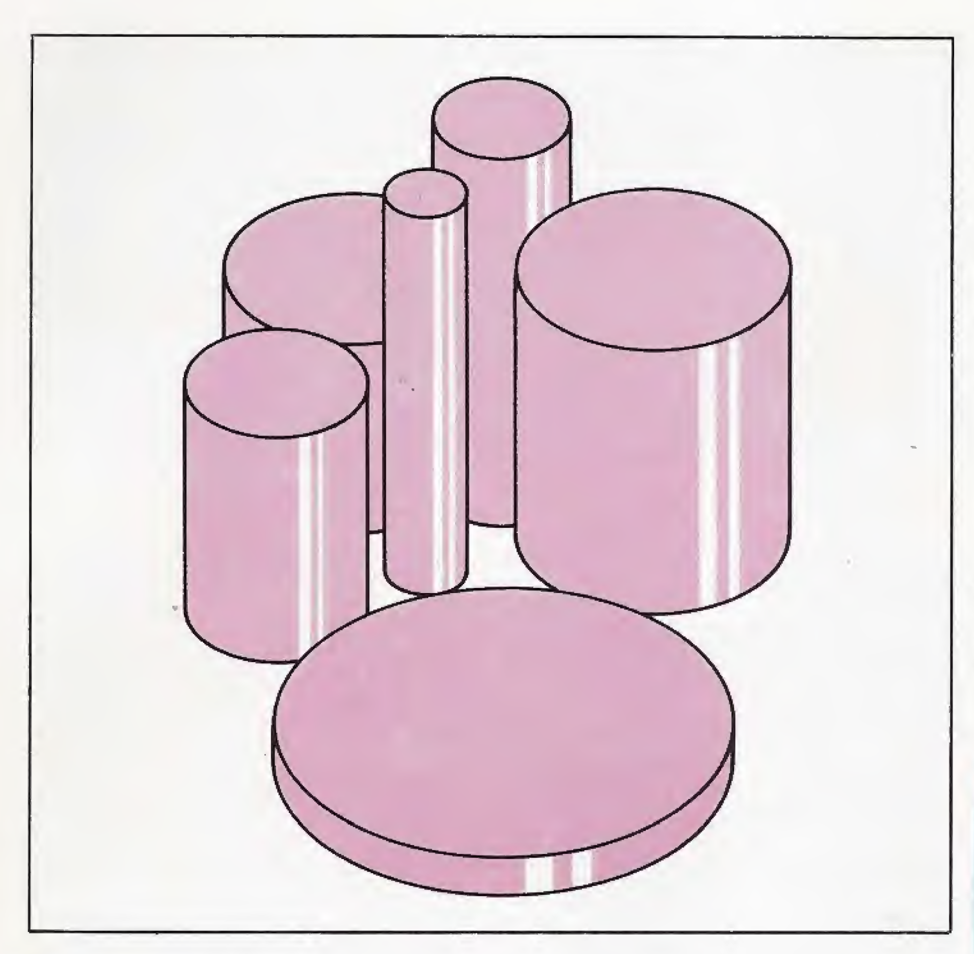
Ha llegado el momento de introducir una nueva posibilidad habitual en el BASIC. Esta es la de incluir más de una instrucción por línea de programa. En efecto, en una misma línea pueden coexistir varias instrucciones. Para ello, hay



El problema de los botes de conserva... Un problema de optimización en el que el ordenador muestra su amplia capacidad para los cálculos reiterativos.



En definitiva, el ejemplo de los botes equivale a un proceso de optimización. ¿Qué dimensiones ha de tener el bote para reducir al mínimo su coste, para un volumen dado?



¿Qué formato es el más económico? La decisión supone acometer toda una serie de operaciones repetitivas que el ordenador ha de ejecutar.

que separarlas entre sí por medio del signo «dos puntos» (:).

Ahora, las dos últimas líneas del ejemplo pueden agruparse en una sola:

```
40 K=K+25:GOTO 20
```

Ello supone un ahorro en líneas de programa. En todo caso, el mayor interés de la coexistencia de varias instrucciones en una misma línea, radica en las instrucciones del tipo IF/THEN. En el ejemplo anterior, se describían en la pantalla todos los valores localizados por encima del buscado. Haciendo uso de esta nueva posibilidad, puede lograrse que sólo aparezca en la pantalla el valor crítico: número de kilómetros a partir de los que la segunda fórmula de alquiler resulta más cara que la primera.

```
10 K=50
20 P2=3000+15*K
30 IF P2>8000 THEN PRINT K:STOP
40 K=K+25
50 GOTO 20
```

Una vez que se determina el valor en cuestión, se detiene la ejecución del programa. También es posible definir una presentación más atractiva, mostrando los kilómetros y el precio.

```
10 K=50
20 P2=3000+15*K
30 IF P2>8000 THEN PRINT K;"KM A"; P2;"PTAS":STOP
40 K=K+25
50 GOTO 20
```

Otra alternativa consiste en invertir la condición impuesta en la instrucción IF. Si bien, para no alterar el resultado, habrá que intercambiar las instrucciones de la zona THEN con las incluidas en las líneas 40 y 50. Este será el nuevo aspecto del programa:

```
10 K=50
20 P2=3000+15*K
30 IF P2>8000 THEN K=K+25:GOTO 20
40 PRINT K;" KM A ";P2" PTAS"
50 STOP
```

En cualquiera de los dos últimos programas puede precisarse con mayor exactitud el resultado, cambiando el valor del incremento de 25 a 1 Km. Para ello, bastará con sustituir la asignación K=K+25 por K=K+1. En tal caso, la ejecución conducirá al siguiente resultado:

```
RUN<CR>
334 KM A 8010 PTAS
█
```

El problema de los botes de conserva

El segundo ejemplo se ocupará de acometer un cálculo de optimización. El problema consiste en diseñar un recipiente que, con el mismo contenido, utilice la mínima cantidad de material. Se partirá de un bote cilíndrico, cuyo volumen es de medio litro, al que se le variarán las dimensiones de altura y anchura.

Para medir la cantidad de material a emplear, hay que centrar la atención en la superficie del bote. Esta puede referirse a dos magnitudes: altura y radio de la base. La superficie total será la suma

de las dos bases más el área lateral. Cada base tiene un área PI*R^2, siendo R el radio. A su vez, la superficie lateral será igual a la longitud de la circunferencia de la base multiplicada por la altura del bote. En definitiva, y dado que el perímetro es igual a 2*PI*R, la superficie lateral coincidirá con H*2*PI*R, llamando H a la altura del bote.

Están ya definidos todos los ingredientes cuya suma determinará la superficie total del recipiente:

```
S=2*(PI*R↑2)+H*2*PI*R
```

Esta fórmula se puede reducir sacando el factor común a 2*PI*R, de donde resultará:

```
S=2*PI*R(R+H)
```

Lo único que hace falta es determinar los valores de R y H. Estos son los que han de variarse para encontrar la solución. Si bien, como quiera que el volumen del bote es fijo (medio litro), ambos estarán relacionados. La fórmula que relaciona el radio y la altura es precisamente la que proporciona el volumen:

```
V=PI*R↑2*H
```

Dado que el volumen es un dato fijo, lo que conviene es aislar la altura en función del radio (o viceversa). Tomando ya el valor de V igual a 500 cc. (medio litro) la expresión final será la que sigue:

```
H=500/(PI*R↑2)
```

De esta forma, variando únicamente el radio, se puede determinar la altura y la superficie del recipiente. Tras esta amplia disertación matemática, se llega finalmente al programa:

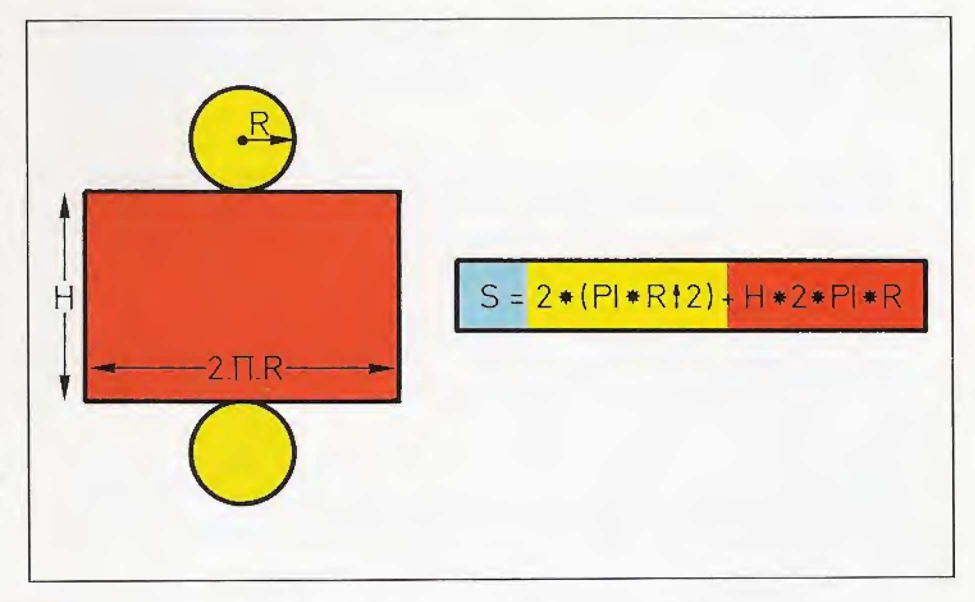
```
20 PRINT "RADIO","ALTURA","SUPERFICIE"
30 PI=3.1416
40 FOR R=1 TO 10 STEP 0.5
50 H=500/(PI*R↑2)
60 S=2*PI*R*(R+H)
70 PRINT R,H,S
80 NEXT R
90 END
```

Un vez completado el programa puede ya introducirse la orden RUN para que sea ejecutado por el ordenador. Este será el resultado.

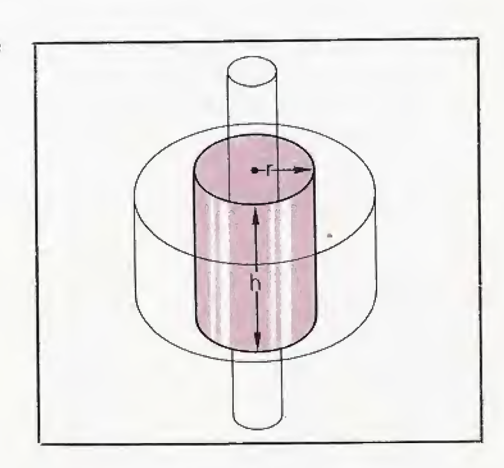
RADIO	ALTURA	SUPERFICIE
1	159.155	1006.28
1.5	70.7354	680.804
2	39.7886	525.133
2.5	25.4647	439.27
3	17.6838	389.882
3.5	12.9922	362.683
4	9.94716	350.531
4.5	7.85948	349.457
5	6.36618	357.08
5.5	5.26131	371.885
6	4.42096	392.862
6.5	3.76697	419.311
7	3.24805	450.734
7.5	2.82941	486.763
8	2.48679	527.125
8.5	2.20283	571.608
9	1.96487	620.05
9.5	1.76349	672.322
10	1.59155	728.32

A la vista de la tabla, se observa de inmediato que la menor superficie se encuentra entre los valores del radio comprendido entre 4 y 5 cm. Para hallar un valor más exacto construiremos un nuevo programa. En esencia, coincidirá con el anterior con la única salvedad que supone cambiar los valores límite y el incremento. El margen a inspeccionar es el situado entre los valores del radio delimitados por 4 y 5 cm. En consecuencia, éstos serán los límites del bucle FOR. El incremento se reducirá, asimismo, a 0.05 en la zona STEP de la instrucción FOR:

```
20 PRINT "RADIO","ALTURA","SUPERFICIE"
30 PI=3.1416
40 FOR R=4 TO 5 STEP 0.05
```



El área total del bote se calcula a partir de dos datos: radio y altura. La resolución del problema pasa por comparar los distintos valores de superficie para determinar cuál es la mínima.



Para llegar a establecer las dimensiones óptimas hay que variar en los sucesivos cálculos las medidas del radio (R) y la altura (H) del bote.

```
VOLUMEN?500
RADIO MINIMO?4
RADIO MAXIMO?5
```

RADIO	ALTURA	SUPERFICIE
4	9.947716	350.531
4.05	9.70307	349.974
4.1	9.46785	349.523
4.15	9.24108	349.176
4.2	9.02237	348.931
4.25	8.81132	348.784
4.3	8.6067	348.734
4.35	8.41086	348.779
4.4	8.22079	348.915
4.45	8.03709	349.142
4.5	7.85948	349.457
4.55	7.68769	349.858
4.6	7.52147	350.344
4.65	7.37059	350.912
4.7	7.20482	351.562
4.75	7.05393	352.291
4.8	6.90774	353.098
4.85	6.76605	353.982
4.9	6.62867	354.941
4.95	6.49543	355.974
5	3.36617	357.08

```
SOLUCION OPTIMA:
RADIO=4.3
ALTURA=8.6076
SUPERFICIE=348.734
```

```
50 H  500/(PI*R↑2)
60 S=2*PI*R*(R+H)
70 PRINT R,H,S
80 NEXT R
90 END
```

Desde luego, el programa puede también encargarse de decidir cuál es la solución óptima. Para ello, debe encontrar el valor mínimo de la superficie. Dicho valor puede determinarlo comparando sucesivamente los contenidos da la variable S. En orden a facilitar su tarea, es preciso que entren en escena otras dos variables SMIN y RMIN. En la primera se guardará el valor mínimo de S resultante de las comparaciones parciales. A su vez, RMIN almacenará el radio correspondiente a dicha área mínima.

Ello se traduce en la incorporación de una nueva línea:

```
75 IF S>SMIN THEN SMIN=S:RMIN=R
```

Esta línea irá comparando los valores de S e introduciendo el menor de ellos en SMIN, mientras que en RMIN quedará el radio correspondiente.

El único inconveniente estriba en la primera ejecución del bucle. SMIN ha de tener un valor inicial. Si dicho valor es inferior a todos los de S, nunca llegará a cumplirse la condición S<SMIN. Por lo tanto, hay que extremar el cuidado al elegir el valor inicial de SMIN. Para evitar cualquier problema, se inicializará un valor suficientemente alto, por ejemplo:

```
35 SMIN=50000
```

Por supuesto, al concluir la ejecución hay que mostrar en la pantalla el valor del radio mínimo. De ello se ocuparán algunas instrucciones PRINT.

```
 90 PRINT "SOLUCION OPTIMA:"
100 PRINT "RADIO=";RMIN
110 PRINT "ALTURA=";500/(PI*RMIN↑2)
120 PRINT "SUPERFICIE=";SMIN
```

Aún cabe pulir en mayor grado el programa. Se puede contemplar la posibilidad de alterar el volumen de los botes. Para ello no tomaremos el dato constante 500; en su lugar, se introducirá la variable V. Esta se inicializará por medio de una instrucción INPUT.

```
10 INPUT "VOLUMEN";V
```

También es necesario introducir los valores mínimo y máximo del radio. En caso contrario podría omitirse el radio idóneo al no estar comprendido entre los límites establecidos. Serán necesarias dos nuevas líneas de programa

```
14 INPUT "RADIO MINIMO";A
15 INPUT "RADIO MAXIMO"; B
```

Un vez completo, éste adopta el siguiente aspecto:

```
  1 REM *LOS BOTES*
 10 INPUT "VOLUMEN";V
 14 INPUT "RADIO MINIMO";A
 15 INPUT "RADIO MAXIMO";B
 20 PRINT "RADIO","ALTURA","SUPERFICIE"
 30 PI=3.1416
 35 SMIN=50000
 40 FOR R=A TO B STEP (B-A/20)
 50 H=V(PI*R↑2)
 60 S=2*PI*R*(R+H)
 70 PRINT R,H,S
 75 IF S>SMIN THEN SMIN=S:RMIN=R
 80 NEXT R
 90 PRINT "SOLUCION OPTIMA"
100 PRINT "RADIO=";RMIN
110 PRINT "ALTURA=";V/(PI*RMIN↑2)
120 PRINT "SUPERFICIE"=";SMIN
130 END
```

Cabe observar que tal como se ha señalado en un párrafo anterior, el valor 500 que aparecía en las líneas 50 y 110, se ha sustituido por la variable V. En la línea 40 se han tomado como límites del bucle los valores A y B, introducidos en las líneas 14 y 15. En ese mismo bucle, se ha fijado el incremento en un valor (B−A)/20. Ello hará que en la tabla aparezcan 20 valores igualmente espaciados.

La ejecución del programa conduce al resultado que aparece en el cuadro adjunto.

Otras estructuras de control

Nuevas formas de crear bucles



Los ordenadores son capaces de realizar un mismo trabajo un elevado número de veces. La ventaja de tal posibilidad de reiteración automática es más que apreciable para el usuario. Las máquinas no se cansan, ni abandonan su tarea... a menos que les falte el suministro de energía. ¿Y para qué están las máquinas, sino para liberar al hombre de esos trabajos monótonos y repetitivos?

La estructura WHILE/WEND

Cualquier versión del lenguaje BASIC, incluso las menos potentes, incorporan una serie de instrucciones de control: IF/THEN/ELSE, FOR/NEXT, GOTO,... Algunos traductores BASIC van más lejos, y ofrecen ciertas instrucciones de control alternativas más versátiles y manejables, cuyo estudio se acometerá en el presente capítulo.

La incorporación de estas nuevas sentencias se debe al intento de aproximar un poco más el BASIC a los lenguajes de programación estructurados. Estos basan su eficacia en el uso exclusivo de tres estructuras básicas: secuencial, selectiva y repetitiva, apoyadas en potentes instrucciones propias de los lenguajes estructurados y para las que el BASIC no tiene réplica en la mayor parte de los casos.

Una de ellas, que aún siendo poco frecuente forma parte del repertorio de instrucciones BASIC de muchos ordenadores personales, es la instrucción WHILE/WEND, cuyo formato más general es el siguiente:

<N1.> WHILE <expr.>
<bucle>
<N1.> WEND

Esta instrucción permite la ejecución repetitiva de un conjunto de sentencias englobadas en un bucle, delimitado por las palabras clave WHILE (por delante) y WEND (al final).

Como se recordará, en las instrucciones FOR/NEXT se repetía el bucle un número de veces determinado de antemano por los valores inicial y final de su variable de control. En esta nueva instrucción, el número de veces que se ejecutará el bucle está indeterminado en un principio. El bucle será ejecutado reiteradamente mientras (WHILE) sea cierta la expresión <expr.> que lo controla; esta última será una expresión de tipo booleano. En el instante en el que dicha expresión deje de adoptar el valor lógico 1 (verdadero), el bucle será abando-

Los ordenadores pueden realizar casi cualquier tipo de tarea repetitiva sin acusar el menor cansancio. Ello los convierte en útiles herramientas capaces de liberar al hombre de los trabajos monótonos.



La potencia del ordenador para ejecutar todo tipo de cálculos es una de las facultades que más ha potenciado su implantación actual.

WHILE/WEND

Repetición de un bucle de sentencias, mientras sea cierta la condición que lo controla.

Formato: <N1.> WHILE <expr.> <bucle> WEND

Ejemplo:

```
10 WHILE I<0
20 I=I+1
30 WEND
```

nado, pasando la secuencia de ejecución del programa a la instrucción situada inmediatamente detrás de la palabra WEND.

El ejemplo que sigue contribuirá a clarificar la diferencia existente entre las instrucciones WHILE/WEND y FOR/NEXT:

```
100 CONTROL=1
110 WHILE CONTROL
120 CONTROL=0
130 FOR I=1 TO 19
140 IF A$(I)>A$(I+1) THEN SWAP
    A$(I), A$(I+1): CONTROL=1
150 NEXT I
160 WEND
█
```

El objetivo es clasificar en orden creciente los valores que se suponen ya almacenados en una matriz de caracteres A$, previamente dimensionada con 20 elementos. El bucle FOR del ejemplo se ejecutará 19 veces, comprobando si un elemento es menor que el siguiente, en cuyo caso se procederá a intercambiar sus posiciones respectivas mediante el comando SWAP. En el bucle WHILE, en cambio, no está determinado en principio el número de veces que se ejecutará; la reiteración proseguirá hasta que estén ordenados todos los elementos de la matriz A$.

La variable que señalará el final de la repetición del bucle es la denominada CONTROL. Su valor lógico se pone a uno (verdadero) cada vez que se efectúa un cambio de orden, para así volver a comprobar si la nueva sucesión de valores de A$ es ya correcta. Como se deduce fácilmente, es necesario modificar el valor de la expresión de control asociada a WHILE dentro de su propio bucle. De no ser así, el bucle no se ejecutaría nunca si la condición impuesta resultara falsa al llegar al mismo. Aunque también podría suceder lo contrario: la ejecución podría eternizarse en un bucle infinito, si la condición resultara cierta al entrar en el bucle WHILE. En el siguiente ejemplo jamás podrá abandonarse la ejecución repetida del bucle WHILE/WEND, puesto que la condición no es actualizada dentro del propio bucle:

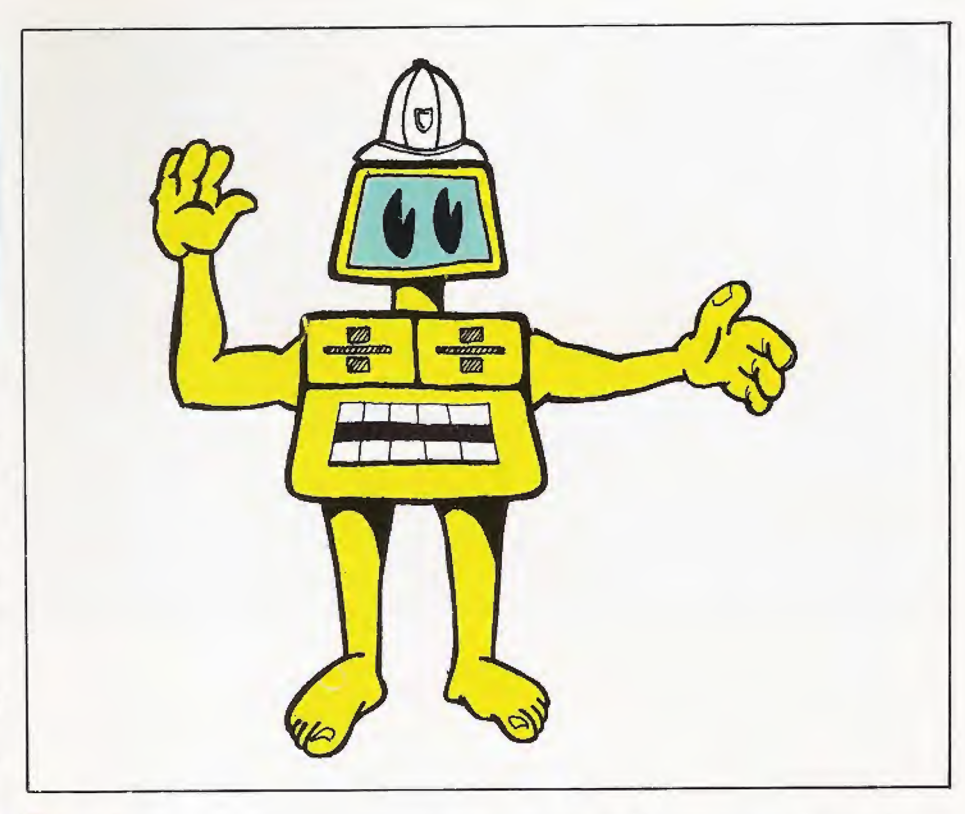
```
200 INDI=1
210 WHILE INDI
220 PRINT "PULSA BREAK, POR FAVOR!"
230 WEND
█
```

De ahí que en el primer ejemplo se asignara el valor 0 a la variable CONTROL en la línea 120: dentro del bucle. Con ello será posible salir del mismo. El valor de CONTROL sólo tomará el contenido lógico 1 —en orden a repetir el bucle—, cuando se efectúe un cambio en el orden de dos o más elementos en la línea 140.

Los bucles WHILE/WEND pueden ser anidados unos dentro de otros, de forma similar a como se anidan los bucles FOR/NEXT o las sentencias IF/THEN/ELSE. En este caso, cada palabra WEND cerrará el bucle correspondiente a la instrucción WHILE que se haya ejecutado con mayor inmediatez. Así, el último WEND corresponderá a la setencia WHILE en cuyo bucle estén inmersas todas las restantes sentencias WHILE anidadas. Otra característica importante de la sentencia WHILE/WEND es que el bucle no se ejecutará ninguna vez si al llegar a ella la expresión que lo controla tiene el valor lógico 0 (falso). El motivo hay que buscarlo en el hecho de que la condición se evalúa antes de entrar en el bucle, y no tras entrar en el mismo, en cuyo caso siempre se ejecutaría el bucle al menos una vez.

La estructura REPEAT/UNTIL

Otra estructura de control, también presente en varios intérpretes BASIC, aunque menos habitual que la WHILE/WEND, es REPEAT/UNTIL. Raro es el traductor BASIC que posee ambas estructuras, de tal forma que si tiene una de ellas, la otra, generalmente, estará ausente. Su formato, muy similar al de



El BASIC es un celoso guardián de la secuencia de ejecución de un programa, dirigiendo su flujo mediante las instrucciones de control.

la estructura WHILE/WEND, es el siguiente:

<N1.> REPEAT

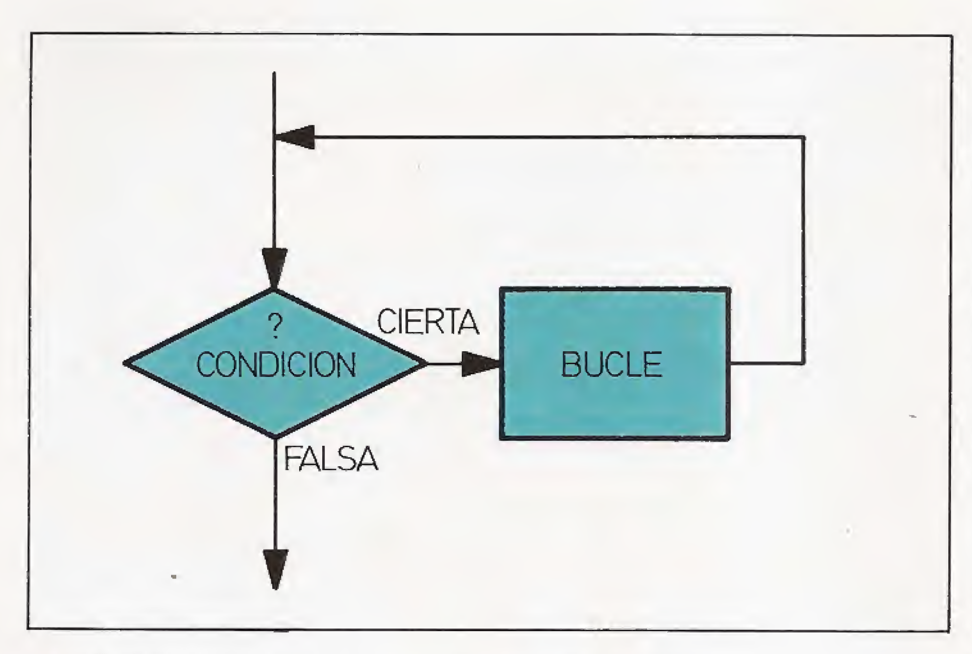
<bucle>

<N1.> UNTIL <expr.>

En este caso, el bucle queda delimitado por las palabras clave REPEAT (comienzo) y UNTIL (final). A continuación de esta última figura la expresión <expr.>, que determinará el instante en que se debe abandonar la ejecución del bucle.



La instrucción WHILE hará que se repita una determinada acción mientras que sea cierta la condición impuesta.



WHILE da entrada a una estructura de control en la que la condición se evalúa antes de ejecutar el bucle asociado.

La principal diferencia con la estructura WHILE/WEND, en cuanto a su funcionamiento, es que ahora el conjunto de sentencias que forman el bucle se ejecutará siempre al menos una vez, sea cual sea el valor lógico de la expresión. Al contrario de la estructura WHILE/WEND, la condición se evalúa al llegar al final del bucle; de forma que si tiene el valor lógico 0 (falso), se volverá a repetir el bucle. Así sucesivamente hasta que la expresión de control adopte el valor lógico 1 (verdadero), en cuyo caso el programa seguirá ejecutándose a partir de la instrucción cuyo número de línea sigue al que corresponde a la línea que contiene la palabra UNTIL.

Un ejemplo de aplicación de esta estructura cabe situarlo en un programa de juego, en el que la ejecución del mismo no comienza hasta que se haya pulsado una tecla cualquiera. Para ello, se explorará el teclado repetidamente, mediante el comando INKEY$, hasta que se detecte la pulsación de alguna tecla. Esta simple instrucción conformará el bucle de la estructura iterativa.

La condición a evaluar cada vez que se ejecute el bucle es: ¿se ha pulsado ya alguna tecla? En caso afirmativo, se abandonará el bucle y se ejecutará el resto del programa. Al efecto se puede utilizar una variable inicializada con el valor lógico 0 (falso), pasando a asignarle el valor 1 (verdadero) en el momento en el que se cumpla la condición impuesta.

Esta se verificará al pulsar una tecla, lo que se detectará cuando el valor obtenido por INKEY$ sea distinto de "" (cadena vacía) que es el valor normal de vuelta cuando no se ha accionado tecla alguna.

El listado del programa capaz de realizar tal cometido puede coincidir con el siguiente:

```
10 PRINT "PULSE UNA TECLA CUALQUIERA PARA EMPE
   ZAR"
20 YAESTA=0
30 REPEAT
40 IF INKEY$<>"" THEN YAESTA=1
50 UNTIL YAESTA
60 REM AQUI EMPIEZA EL PROGRAMA PRINCIPAL
```

REPEAT/UNTIL

Repetición de un bucle de sentencias, hasta que sea cierta la condición que lo controla.

Formato: <N1.> REPEAT <bucle> UNTIL <expr.>

Ejemplo:
```
10 REPEAT
20 N=N+1
30 PRINT N
40 UNTIL N=10
```

La estructura REPEAT evalúa la condición establecida al finalizar la ejecución del bucle. Ello significa que éste se ejecutará al menos una vez.

La variable YAESTA será la que indique el momento en el que debe ser abandonado el bucle: cuando su valor lógico sea 1. De inmediato, se pasará a ejecutar la línea 60 en la que dará comienzo el programa de juego.

Dado que esta estructura permite que se ejecute el bucle al menos una vez, no hay peligro de que el ordenador quede encerrado en un bucle sin salida al asignar a YAESTA el valor lógico 0 en la línea 20 del bucle.



El BASIC de COMMODORE es uno de los pocos que incorpora la instrucción de control DO/LOOP.

Más y más lazos

Una sentencia BASIC muy similar a WHILE/WEND y REPEAT/UNTIL, aunque menos frecuente aún en el variado repertorio de intérpretes BASIC, es la instrucción DO/LOOP. Esta puede considerarse como un compendio de la WHILE/WEND y de la REPEAT/UNTIL, por el motivo que se verá más adelante.

Su formato es el siguiente:

```
<N1.> DO[WHILE <expr.>]
<bucle>
<N1.> LOOP [UNTIL <expr.>]
```

Ahora, la rutina que constituye el bucle está delimitada por las palabras clave DO como principio y LOOP como final. La salida del bucle se controla, al igual que en los casos anteriores, por medio del valor lógico que toma una determinada expresión. Hasta aquí no hay novedad alguna.

La diferencia radica en que mientras en la estructura WHILE/WEND la condición para abandonar el bucle se evalúa al principio, y en la REPEAT/UNTIL al final del mismo, en la estructura DO/LOOP la condición puede ser chequeada en cualquiera de los dos puntos, o incluso en ambos. Por esta razón se afirmaba anteriormente que DO/LOOP puede considerarse como una extensión de las otras dos estructuras, lo que le confiere una mayor potencia. La expresión que con su valor lógico determina cuándo debe ser abandonado el bucle, puede ir situada como argumento de una función WHILE (mientras que...), o bien como argumento de una función UNTIL (hasta que...). Estas pueden emplazarse detrás de la palabra DO que abre el bucle, o detrás de la palabra LOOP que lo cierra, o incluso en ambas. De esta forma se conseguirá salir del bucle por cualquiera de los dos puntos, dependiendo de los valores lógicos de ambas condiciones.

Esta característica resulta de gran utilidad para evaluar a la vez dos condiciones distintas, de forma que al cumplirse una cualquiera de ellas se abandone la repetición del bucle.

```
 5 INPUT NUM
10 MIN=10:MAX=50
20 DATA 15,13,—20,5—11,18,25,—15,4,1000
30 DO WHILE NUM>MIN
40 READ D
50 PRINT NUM
60 NUM=NUM+D
70 LOOP UNTIL NUM>MAX
80 PRINT "FIN"
```

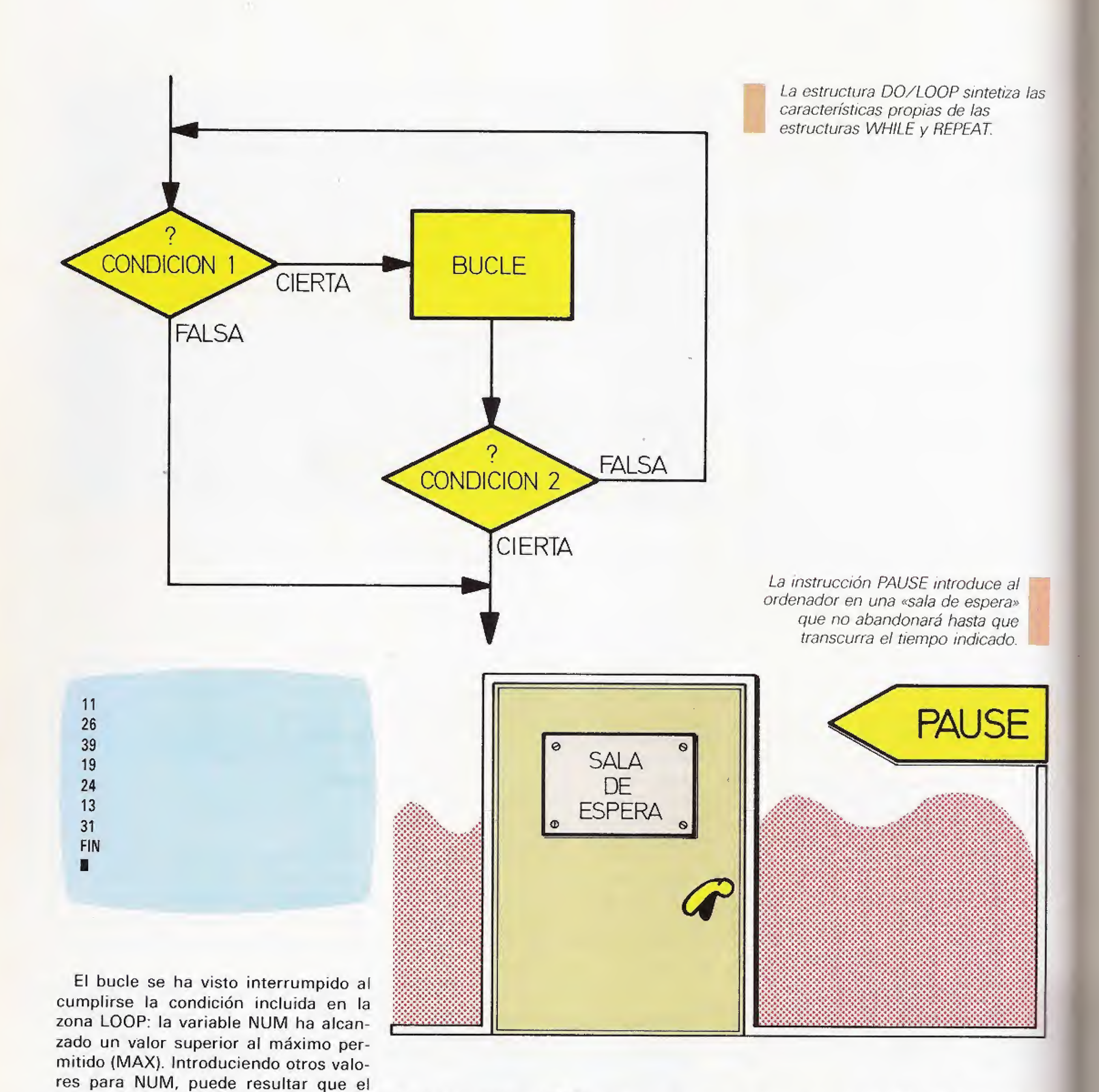
La ejecución del anterior, al introducir por el teclado el valor 11 como asignación de NUM, mostrará en la pantalla los siguientes valores aleatorios comprendidos entre 10 y 50:

DO/LOOP

Repetición de un bucle de instrucciones, mientras o hasta que sean ciertas las condiciones que lo controlan.

Formato:
```
<N1.> DO (WHILE <expr. 1>) <bucle> LOOP (UNTIL <expr. 2>)
```

Ejemplo:
```
10 DO WHILE FLIP <> FLOP
20 IF FLIP>FLOP THEN FLIP=FLIP—FLOP
   ELSE FLOP=FLOP—FLIP
30 LOOP UNTIL FLOP>FLIP
```



La estructura DO/LOOP sintetiza las características propias de las estructuras WHILE y REPEAT.

La instrucción PAUSE introduce al ordenador en una «sala de espera» que no abandonará hasta que transcurra el tiempo indicado.

El bucle se ha visto interrumpido al cumplirse la condición incluida en la zona LOOP: la variable NUM ha alcanzado un valor superior al máximo permitido (MAX). Introduciendo otros valores para NUM, puede resultar que el lazo se interrumpa al verificarse la condición de la zona DO.

Generación de retardos en BASIC

El empleo de retardos es tan útil en multitud de ocasiones, que la mayoría de los intérpretes BASIC poseen un comando específico cuya única misión es exclusivamente ésa. La forma más general de este comando es:

PAUSE <expr.>

El valor de <expr.> representa el tiempo que el ordenador se detendrá antes de continuar con su tarea. Puesto que depende del reloj interno del ordenador, este valor será muy diferente de unos ordenadores a otros. Así, por ejemplo, para obtener una pausa de 1 segundo, en unos casos habrá que hacer PAUSA 1, mientras que en otros será necesario poner PAUSA 60.

TABLA DE CONVERSION				
Ordenador	WHILE/WEND	REPEAT/UNTIL	DO/LOOP	PAUSE
	WHILE <e1>: WEND	REPEAT: UNTIL <e1>	DO WHILE <e1> LOOP UNTIL <e2>	PAUSE <exp.>
AMSTRAD	WHILE <e1>: WEND	—	—	—
APPLE II (APPLESOFT)	—	—	—	
APRICOT (M-BASIC) (M-BASIC) —	—	—	—	PAUSE <exp.>
ATARI	—	—	—	—
CMB 64	—	—	—	—
DRAGON	—	—	—	—
EQUIPOS MSX	—	—	—	PAUSE <exp.>
HP-150	WHILE <e1>: WEND	—	—	—
IBM PC	WHILE <e1>: WEND	—	—	PAUSE <exp.>
MPF	—	—	—	—
NCR DM-V (MS-BASIC)	WHILE <e1>: WEND	—	—	—
NEW BRAIN	—	—	—	—
ORIC	—	REPEAT: UNTIL <e1>	—	—
SHARP MZ-700 (MZ-BASIC)	—	—	—	—
SINCLAIR QL	—	REPEAT: ENDREPEAT <e1>	—	PAUSE <exp.>
SPECTRAVIDEO	—	—	—	—
ZX-SPECTRUM	—	—	—	PAUSE <exp.>

PAUSE

Detención temporal de la ejecución de un programa durante un tiempo especificado.

Formato: <N1.> PAUSE <expr.>

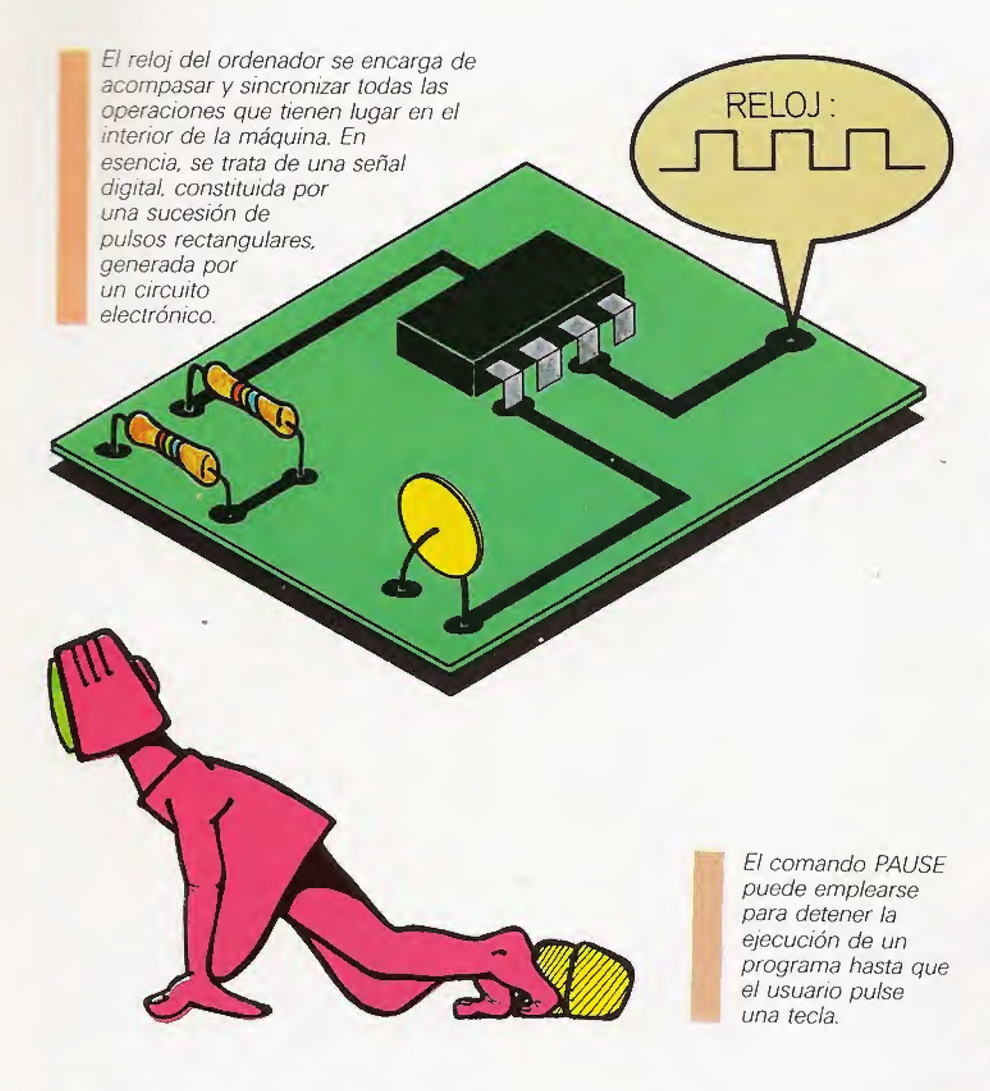
Ejemplos: 10 PAUSE 60
100 PAUSE N+10

Una vez determinado el valor adecuado para obtener una pausa de un segundo, el cálculo del valor necesario para cualquier otro retardo se reduce a una simple multiplicación.

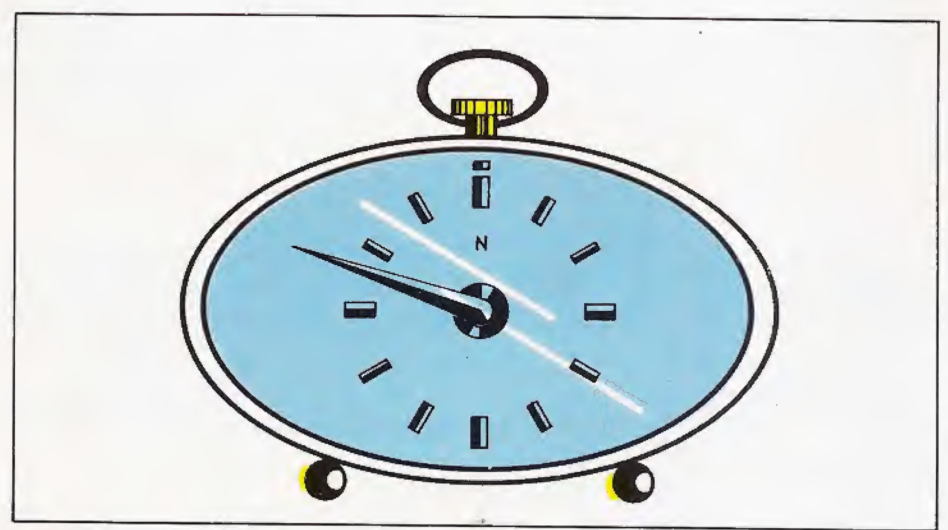
El siguiente ejemplo ilustra el empleo de la instrucción PAUSE:

```
5 PI=3.14159
10 FOR H=1 TO 12
20 PRINT AT 20+9*SIN(H/6*PI), 11—9*COS(H/6*PI);H
30 NEXT H
40 FOR S=0 TO 10000
50 LET G=S/30*PI
```

El reloj del ordenador se encarga de acompasar y sincronizar todas las operaciones que tienen lugar en el interior de la máquina. En esencia, se trata de una señal digital, constituida por una sucesión de pulsos rectangulares, generada por un circuito electrónico.



El comando PAUSE puede emplearse para detener la ejecución de un programa hasta que el usuario pulse una tecla.



Otra de las utilidades de PAUSE es la generación de bucles temporales. Una aplicación clara es la conversión del ordenador en un eficaz cronómetro.

```
 60 LET X=20+10*SIN(G)
 70 LET Y=11—10*COS(G)
 80 PRINT AT X, Y;"."
 90 PAUSE 60
100 PRINT AT X, Y;" "
110 NEXT S
```

Al ejecutar el programa, aparecerá en la pantalla el siguiente resultado:



En efecto, el programa tiene como objeto dibujar en la pantalla un reloj que mida los segundos. Su correcto funcionamiento en distintos ordenadores se verá limitado por el hecho, comentado anteriormente, de los distintos valores aplicables a la sentencia PAUSE.

El valor de PAUSE de la línea 90 debe ser modificado en cada caso concreto para hacer que el reloj evolucione correctamente. Asimismo, será necesario en ciertos casos modificar la escala de las posiciones en pantalla para la sentencia PRINT AT, así como la inclusión de la constante PI (3,1416).

Hay que tener en cuenta el tiempo invertido en realizar el cálculo de los valores de G, y el tiempo empleado en imprimir en la pantalla el signo ".", y borrarlo. Ello significa que el tiempo muerto consumido por la sentencia PAUSE no debe ser exactamente de un segundo si se quiere obtener un reloj de apreciable exactitud. Será preciso cronometrarlo con un reloj convencional, e ir ajustando el valor PAUSE hasta su valor correcto.

Otra característica de la sentencia PAUSE es que puede inhibirse al pulsar una tecla cualquiera del teclado. Con ello se puede lograr el efecto de arrancar un programa al cabo de un tiempo determinado, o bien en el instante en el que alguien accione una tecla cualquiera.

Tipos de variables

Representación de datos en BASIC



Las magnitudes medibles se expresan en unidades. Así, para longitud se emplea el metro, para masa el gramo, etc. Sin embargo, es corriente acudir a múltiplos o submúltiplos cuando las cantidades son muy grandes o muy pequeñas. Esto es lo que sucede, por ejemplo, con los metros. Nadie indicaría la distancia que separa Madrid de la Coruña en metros. Por lo general estas distancias grandes se suelen expresar en kilómetros. De la misma forma, el tamaño de una moneda se da en centímetros o incluso en milímetros. En aplicaciones matemáticas, y científicas en general, se emplean frecuentemente cantidades muy pequeñas o muy grandes. En ese caso se recurre a una representación exponencial, evitándose de este modo la necesidad de escribir un gran número de cifras. Veamos un ejemplo:

$12000000 = 1{,}2 \times 10^{7}$
$0{,}000002 = 2 \times 10^{-6}$

Como se observa, la representación exponencial evita la exigencia de escribir una larga ristra de cifras (especialmente cuando éstas no son más que ceros). Además puede suceder que aunque las restantes cifras no coincidan con el cero, se opte por realizar una aproximación. Ello será válido cuando el error cometido no sea significativo. Por ejemplo:

$1000002 \simeq 10^{6}$

Desde luego, con esta aproximación se comete un error. De todas formas, a veces es más práctico trabajar con aproximaciones si el error puede considerarse despreciable o mínimo.

Una de las posibilidades más fáciles de explotar en un ordenador es la capacidad que éste tiene para realizar cálculos numéricos. Los datos pertinentes, o bien se encuentran ya incluidos en el propio programa o son introducidos durante la ejecución del mismo.

En cualquier cálculo realizado con el ordenador hay que considerar dos factores básicos: los datos que servirán de materia prima y la forma en la que el ordenador los almacena. Todo ello influirá directamente en la ejecución del cálculo.



Una de las facultades básicas del ordenador reside en su aptitud para realizar cálculos numéricos. Los datos a operar son suministrados por el propio programa, o son recogidos del exterior a través de los oportunos dispositivos periféricos.

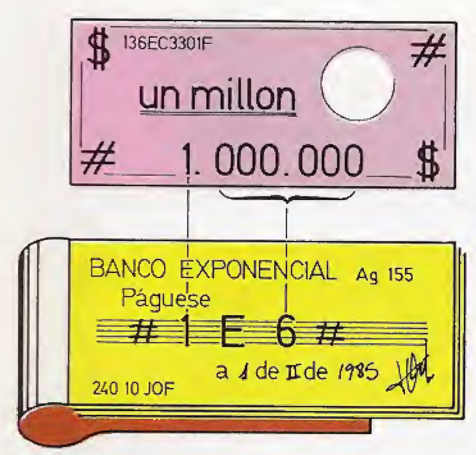
Es evidente que la máquina ha de guardar los datos para más tarde operar con ellos. El lugar en el que almacena o «anota» los datos en cuestión no es otro que la memoria; en ella almacena tanto los datos iniciales como los resultados obtenidos.

En este sentido, hay que tener en cuenta que la memoria del ordenador es limitada, de ahí que, en muchos casos, el usuario se vea obligado a economizarla en la medida de lo posible.

Por otra parte, los datos con los que se ha de operar no son siempre semejantes: pueden incluir cifras decimales en mayor o menor cantidad, pueden representar magnitudes muy elevadas o muy reducidas... Estos son criterios que harán necesario un distinto espacio de memoria para su almacenamiento.



La diversidad de los datos que debe procesar el ordenador supone precisar distintos formatos para su representación interna y almacenamiento en memoria de la máquina.



El formato exponencial permite expresar números de magnitud extrema, muy grandes o muy pequeños, sin tener por ello que utilizar una ingente cantidad de cifras.

Dado que el ordenador no puede conocer a priori la longitud de un número, estos deben almacenarse bajo algún formato estándar. De este modo, es evidente que los datos de poca longitud desaprovecharán un considerable espacio de memoria. Para solventar estos inconvenientes, el BASIC admite distintos tipos de formatos para el almacenamiento de los datos.

Tipos de representaciones numéricas

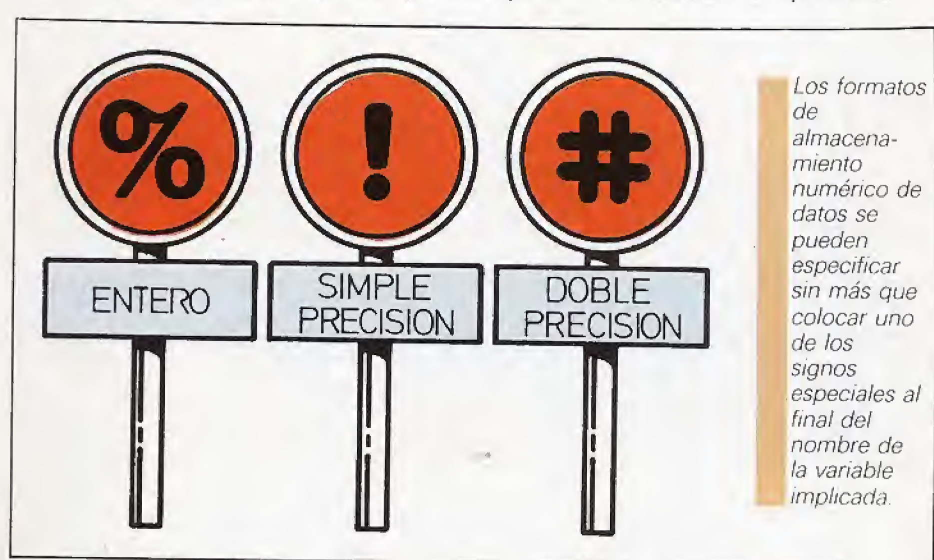
El primer caso es aquel en el que el ordenador ha de almacenar un número entero. Para ello se recurre a una representación binaria pura, empleando a tal fin dos bytes de memoria. Teniendo en cuenta que el primer bit se emplea para el signo, el mayor número representable es el 65535.

Aumentando el número de bytes utilizados, se puede conseguir la representación de números mayores. Sin embargo, este método no resulta práctico desde el punto de vista del aprovechamiento idóneo de la memoria del ordenador. En definitiva, o bien se dispone de una gran variedad de tipos de formato o no quedará más alternativa que despilfarrar espacio de memoria cuando haya que almacenar números de reducida magnitud.

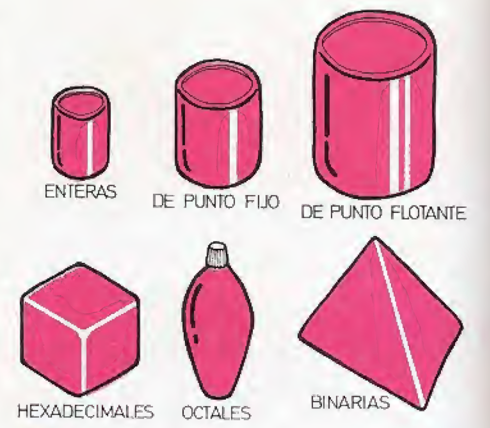
También queda pendiente el caso de los números con parte fraccionaria. ¿Cómo representar este tipo de datos?

Una posible solución reside en adoptar un convenio y fijar la posición de la coma decimal. De esta forma, el número de decimales queda fijado también. Esta representación recibe el nombre de «punto fijo» (o coma fija).

Las limitaciones que presenta la representación en punto fijo pueden ser solventadas mediante un nuevo sistema. Este consiste en utilizar la notación científica de la que se habló al principio. Dicha notación consiste en dividir el número en dos partes. Una de ellas coincide con los dígitos más significativos, mientras que la otra da una idea del orden de magnitud de la cantidad, expresada como potencia de 10. El producto de ambas partes dará un resultado equivalente al número a representar.



Los formatos de almacenamiento numérico de datos se pueden especificar sin más que colocar uno de los signos especiales al final del nombre de la variable implicada.



En función del tipo y tamaño de la constante, el ordenador la almacenará adoptando uno u otro formato entre el repetorio de admisibles.

En el interior de la máquina, estas dos partes se almacenan como «mantisa» y «exponente». La primera aporta los dígitos más representativos, en coma fija. La segunda guarda la potencia de 10 pertinente, a modo de número entero. Esta representación es denominada «punto (o coma) flotante».

Habitualmente, existen dos tipos de representación en coma flotante: simple y doble precisión.

En simple precisión se utilizan cuatro bytes: los tres primeros destinados a la mantisa y el cuarto al exponente. Los bits situados más a la izquierda en cada una de las zonas indican el signo.

La representación en doble precisión emplea siete bytes para la mantisa, y uno para el exponente. Como en el caso anterior, el bit situado más a la izquierda denota el signo de cada parte.

Formatos de almacenamiento en BASIC

En el lenguaje BASIC, los formatos más habituales para el almacenamiento de datos numéricos son los siguientes:

— *Constantes enteras:* se emplean para representar números enteros comprendidos entre -32768 y 32767. No admiten punto decimal.

Ejemplos: 3, 13500, -20000

— *Constantes de punto fijo:* son números reeales en los que se emplea el punto para separar la parte entera de la

fraccionaria, adoptando, en consecuencia, la notación anglosajona. No admite más signos de puntuación. Es decir, no se emplean las comas para separar las unidades de millar de millón.

Ejemplos: 22.57, 10.0, -12.1, 0.12

— *Constantes de punto flotante:* números positivos o negativos representativos en forma exponencial (notación científica). Constan de un número entero o en punto fijo (mantisa) y un entero (exponente). Ambas zonas se separan por medio de la letra E o D. Los valores del exponente no pueden ser mayores de 38 o inferiores a -38.

Ejemplos: 2E3, 3.5 E—6, 7.08 D 14.

Dentro de las constantes de punto flotante cabe distinguir dos tipos: de simple precisión y de doble precisión.

Una constante de simple precisión es toda aquella que satisface los siguientes requisitos:

- Posee hasta siete dígitos de precisión.
- Adopta la forma exponencial utilizando la letra E.
- Incluye un signo de admiración al final (!).

Constante de doble precisión será, por el contrario, la caracterizada por:

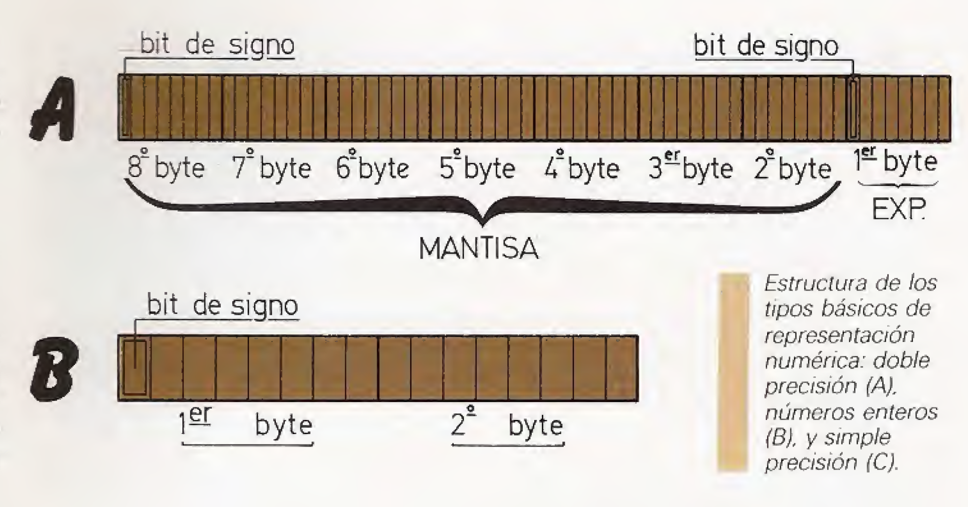
- Más de siete dígitos de precisión.
- Forma exponencial con D.
- Terminada con el signo de número (#).

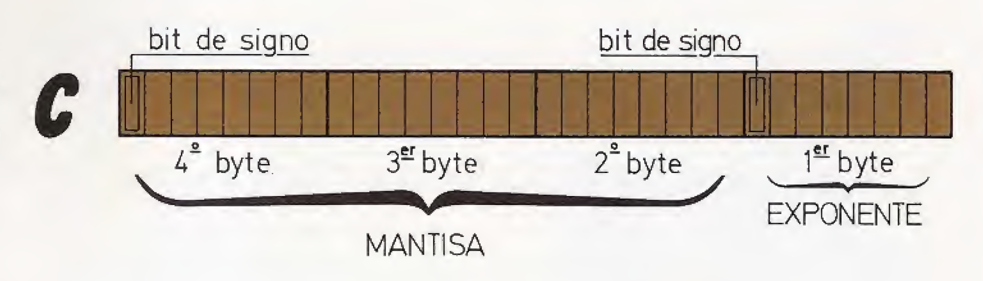
Los números de precisión sencilla se almacenan con siete dígitos de precisión, mostrándose únicamente los seis más significativos, mientras que los de doble precisión se almacenan con dieciséis dígitos.

Otros sistemas de numeración

Desde luego, es posible trabajar con otros sistemas de numeración distintos del decimal (de base 10). Sin ir más lejos, el ordenador utiliza su propio sistema de numeración: el binario o de base 2. Ello se debe a la facilidad que tiene la máquina para manejar «ceros» y «unos» (los únicos dígitos del sistema binario).

Lo que resulta muy cómodo para el ordenador se vuelve claramente incómo-





Estructura de los tipos básicos de representación numérica: doble precisión (A), números enteros (B), y simple precisión (C).

do para el usuario. Así, por ejemplo, el número 8 decimal, se expresa en binario como 1000; análogamente, el número decimal 1000 necesita nueve cifras en su expresión binaria. El problema se agrava por el hecho de que no es inmediata la conversión de binario a decimal, y viceversa.

Los sistemas de numeración más fácilmente transportables al binario son aquellos cuya base es una potencia de 2 (base 4, 8, 16, 32, ...). Entre ellos, los más cercanos a la base 10, y por ende los más utilizados, son el octal (base 8) y el hexadecimal (base 16).

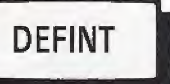
En BASIC se pueden especificar constantes octales y hexadecimales, e incluso operar con ellas. A fin de cuentas, en el interior de la máquina todo se convierte a binario.

— *Constantes hexadecimales:* son números expresados en el sistema de numeración hexadecimal o de base 16. Para su identificación han de ir precedidos por el prefijo &H.

Ejemplos: &H1A45, &H12F

— *Constantes octales:* se trata de números expresados en base ocho (sistema octal). Se distinguen al ir precedidos por &O.

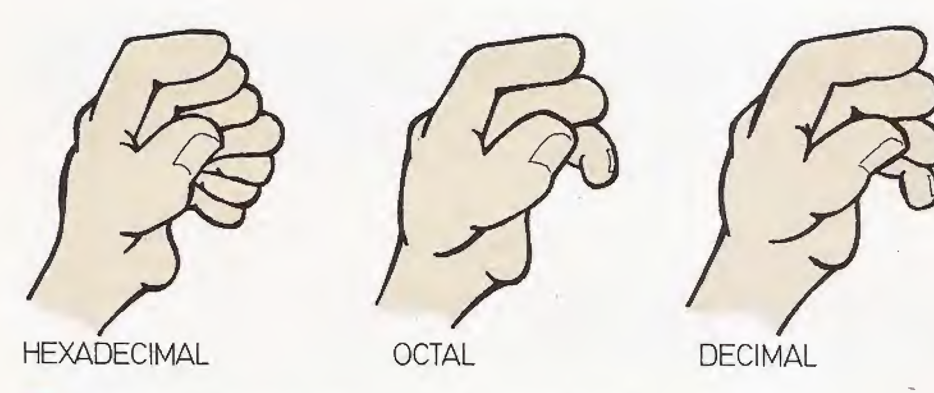
Ejemplo: &O0123, &O264



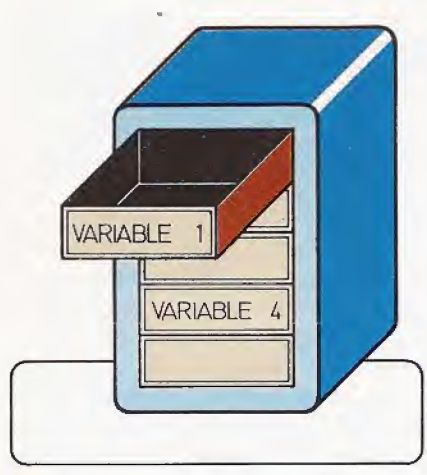
Define las variables cuya inicial se especifica en el formato de números enteros.

FORMATO: DEFINT <inicial>[,<ini1>-<ini2>]...
EJEMPLOS:

```
10 DEFINT H, A—C
50 DEFINT I, L, Z, R
```



Además del decimal hay otros sistemas de numeración perfectamente utilizables para el trabajo práctico. Los dos más difundidos son el octal (base 8) y el hexadecimal (base 16).



Estrictamente, las variables sirven para designar una zona de memoria reservada para acoger valores constantes de un tipo determinado.

Variables

Como ya se ha explicado en capítulos anteriores, una variable puede equipararse a un cajón capaz de contener un dato. Realmente, la variable no es más que un espacio de memoria accesible por medio de un nombre característico: el nombre de la variable. Dicho espacio de memoria puede ver alterado su contenido durante la ejecución del programa. El hecho de que la variable identifique a una zona de memoria específica, implica que deben existir tantos tipos de variables como tipos de datos existan.

De cara al usuario, una variable es un nombre bajo el cual se guarda un contenido. Pero ¿qué restricciones hay en el uso de nombres de variable? Las respuestas difieren según el dialecto BASIC que utilice cada equipo.

La primera limitación está en el número de caracteres permitidos. La mayoría de los ordenadores admiten un número suficiente de caracteres como para identificar con comodidad el contenido de la variable. Así, los nombres VELOCIDAD o GASTOS serán casi siempre válidos. No obstante, otros aparatos sólo reconocen los primeros caracteres, ignorando los restantes. Para un ordenador de este tipo, capaz de reconocer tan sólo los dos primeros caracteres, los siguientes nombres harían referencia a la misma variable:

ALCANCE,ALETAS,AL,ALA,ALTURA

En tal caso, es aconsejable tomar buena nota de los nombres utilizados para no dar a dos variables un mismo identificador.

El nombre de una variable puede incluir, además de letras, números e incluso símbolos y signos de puntuación. Sin embargo, el primer carácter ha de coincidir siempre con una letra.

Un nombre de variable no debe coincidir con ninguna de las palabras reservadas del lenguaje BASIC (comandos o funciones). Si así fuera, el aparato no podría distinguir entre unos y otras. Existen ordenadores que no permiten tan siquiera que una palabra reservada forme parte de un nombre. En ellos no serían válidos los siguientes nombres o variables:

AN*FOR*A, *DIF*ERENCIA, *LET*RAS

Tipos de variables

Una variable pude almacenar un dato de tipo numérico o de tipo alfanumérico (cadena de caracteres). En el segundo caso, el nombre de la variable debe terminar indefectiblemente con el símbolo de dólar. ($).

Dentro de las variables de tipo numérico habrá que distinguir el formato de almacenamiento que utilizan.

En el propio nombre de la variable se puede especificar el formato deseado. Para ello sólo es necesario añadir un símbolo al final, de la misma forma que para las alfanuméricas. Estos símbolos especiales son los siguientes:

% Para variables de tipo entero.
! Para variables de simple precisión.
Para las de doble precisión.

Como se puede observar, estos identificadores de tipo se corresponden con los utilizados para las constantes.

Cuando no se especifica el formato, el

DEFSNG

Define las variables cuya inicial se especifica en el formato de números de simple precisión.

FORMATO: DEFSNG <inicial>[,<ini1>-<ini2>]...
EJEMPLOS:

10 DEFSNG X–Z, B
50 DEFSNG E, L, P

ordenador entenderá que se trata de una variable de simple precisión.

Definición de tipos de datos

Mediante determinadas funciones es posible definir el tipo de dato que almacenarán distintas variables. Sin embargo, en el caso de utilizar ambos métodos de definición, prevalecerá el impuesto por los caracteres indicados en el apartado precedente.

Las funciones de definición permiten elegir el formato de representación interna de una o varias variables. Dicha definición se realiza indicando la inicial de la (o las) variable(s) que utilizarán dicho formato. Las funciones encargadas de este cometido son las siguientes:

DEFINT: Define a las variables correspondientes enteras.

Su formato adopta el siguiente aspecto:

(Número de línea) DEFINIT <letra>, [<letra de comienzo>-<letra final>]...

Mediante esta función se declara que todas las variables que comiencen por la letra indicada serán de tipo entero. Asimismo, también pertenecerán a ese tipo aquellas variables cuya inicial se encuentre entre las letras de comienzo y final. Se pueden especificar tantas definiciones como sea necesario bajo un mismo enunciado; por ejemplo:

```
10 DEFINT I,K—M
```

La definición dada como ejemplo fuerza a que toda variable que comience por las letras I,K,L y M se mantenga en el formato entero.

DEFSNG: Las variables afectadas serán tomadas por el ordenador como variables de simple precisión.

Se rige por el siguiente formato:

(Número de línea) DEFSNG <letra>, [<letra comienzo>-<letra final>]...

El uso de esta función es idéntico al de la anterior, sin más diferencia que el tipo de formato que especifica. Por ejemplo:

DEFDBL

Define las variables cuya inicial se especifica en el formato de números de doble precisión.

FORMATO: DEFDBL <inicial>[,<ini1>-<ini2>]...
EJEMPLOS:

```
10 DEFDBL K, C—G
30 DEFDBL H, E, J
```

```
20 DEFSNG A—F,H
```

En este ejemplo, las variables cuya inicial sea A,B,C,D,E,F o H pertenecerán al tipo de simple precisión.

DEFDBL: Declara a un conjunto de variables como de doble precisión.

Su formato es similar al de las anteriores:

(Número de línea) DEFDBL <letra> [<letra de comienzo>-<letra final>]...

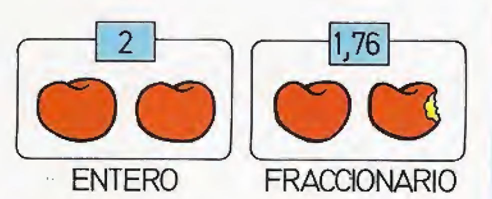
El funcionamiento de DEFDBL es análogo al de las otras dos funciones:

```
30 DEFDBL Z,C—E
```

Ahora, las variables que empiecen por las letras C,D,E y Z serán almacenadas en formato de doble precisión.

Operaciones con distintos tipos de datos

En ocasiones resulta necesario operar con variables y constantes de distintos tipos.



Para poder realizar cálculos correctamente, el BASIC debe poder operar con datos tanto enteros como fraccionarios.

En tal caso, el ordenador reconvierte el tipo de los operandos antes de realizar la respectiva operación. Si se trata de una sentencia de asignación, los datos quedarán convertidos al tipo de la variable receptora del resultado.

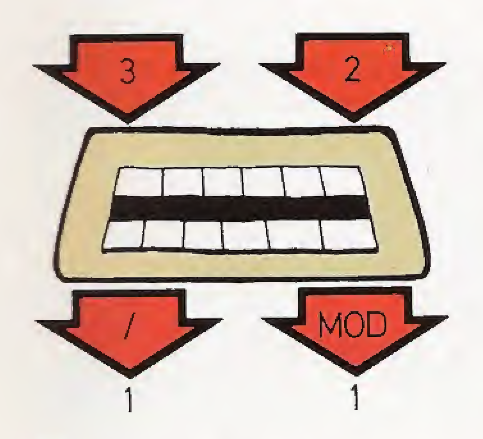
A la hora de evaluar una expresión, todos los operandos se convierten al mismo tipo: el del operando más preciso.

Los operadores lógicos convierten en enteros a sus operandos, generando un resultado entero. Dichos operandos deben tener un valor comprendido entre -32768 y 32767. El resultado de una operación de esta categoría será 0 (falso) ó 1 (verdadero).

Operando con enteros

Cuando se realizan operaciones con datos de tipo entero surge un problema: la división de dos números enteros no siempre da como resultado en número entero. Por ejemplo, si se desea hallar el cociente entre los números enteros 5 y 4 basta con escribir la siguiente línea:





Además de la división convencional, el BASIC dispone de dos funciones auxiliares para operar divisiones de números enteros: la división entera (\) y la función MOD, capaz de obtener el resto entero de un cociente.

El resultado obtenido es a todas luces erróneo. La razón de tan extraño comportamiento debe atribuirse a la naturaleza de los números enteros. Un número entero carece, por definición, de parte fraccionaria. Este hecho implica que el resultado ha de concidir con el entero más próximo. En todo caso, tal eventualidad está prevista en el lenguaje BASIC. La solución llega de la mano de un nuevo operador: el de la división entera. Esta operación permite hallar el cociente entero de la división entre dos números enteros. En otros términos: proporciona el entero más cercano inferior al cociente real.

El símbolo que identifica a esta operación es el denominado «backslash» (\). Su diferencia respecto a la división normal se aprecia a la vista de los siguientes ejemplos:

```
PRINT 5/4 <CR>
1.25
PRINT 4/5 <CR>
.8
█
```

El ordenador ha tomado los datos en formato de simple precisión. Los resultados son números reales, puesto que se ha utilizado el operador de división convencional. Observemos qué ocurre cuando se utiliza la división entera.

```
PRINT 5%\4% <CR>
1
PRINT 4%\5% <CR>
0
█
```

Como era de esperar, los resultados son del tipo entero. Hasta aquí no hay nada sorprendente. En realidad, esta operación parece no ser más que la parte entera del resultado real: ahí va la demostración:

Representación de números negativos

Los ordenadores operan y almacenan los números de acuerdo al sistema de numeración binario. De esta forma, un número se convierte en una serie de unos y ceros, que son más fácilmente manipulables por un circuito electrónico. Este sólo ha de detectar la presencia o ausencia de tensión para así determinar cuál es el valor del dato procesado. La suma de números en el sistema binario no ofrece dificultad alguna, puesto que se reduce a la suma de unos y ceros. En cambio, la resta resulta ser más complicada, sobre todo por la posibilidad de obtener un dato resultante negativo.

La representación de datos negativos en el ordenador, suele solventarse utilizando el bit situado más a la izquierda de la palabra binaria representativa del número en cuestión. Este se emplea como «bit de signo». Si el bit en cuestión se encuentra a «1», el número es negativo; por el contrario, si su valor es «0», el número será positivo (o viceversa, dependiendo del criterio elegido).

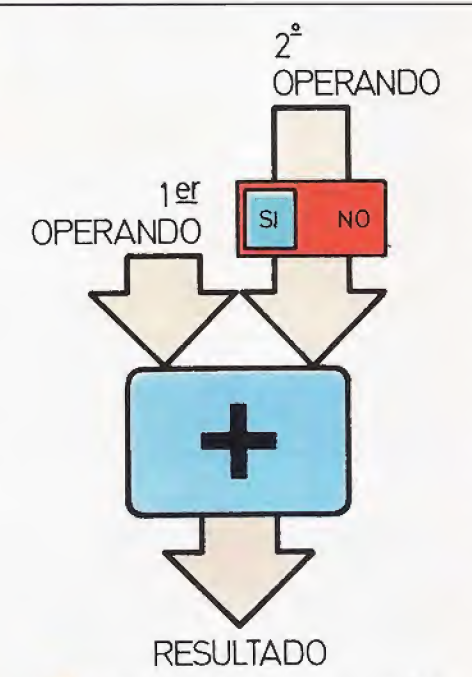
La magnitud del número en cuestión viene determinada por los restantes bits. Estos pueden representar simplemente el valor absoluto del número.

A pesar de todo, este modo de actuación presenta ciertos problemas a la hora de operar. Si se desea realizar la suma de dos datos positivos, el resultado será el correcto. Sin embargo, no ocurre así cuando uno de ambos números es negativo: el resultado coincidirá con un número negativo, de valor igual a la suma de los valores absolutos de los operandos. Tal inconveniente se soluciona con un curioso método de codificación: el denominado «complemento a dos» del número positivo. Este método deriva del complemento a uno habitual, consistente en cambiar unos por ceros y viceversa. Así por ejemplo, el complemento a uno del número 00100011 es 11011100.

Concretamente, el complemento a dos resulta de sumar una unidad al complemento a uno del dato base. Por lo tanto, el complemento a dos del número anterior será 11011101.

Como se observa, el bit de signo ha cambiado, revelando la presencia de un dato negativo. Se puede demostrar que al sumar números de distinto signo, representados de acuerdo a este método, el resultado obtenido coincide con la resta de los operandos.

Esta sencilla técnica permite reducir cualquier sustracción a una operación de suma, sin más que hallar previamente el complemento a dos del sustraendo.



El método del complemento a 2 de un número en expresión binaria facilita la operación de sumar números de distinto signo o, lo que es lo mismo, la obtención de restas.

TABLA DE CONVERSION					
Ordenador	DEFINICION DE TIPO			DIVISION ENTERA	RESTO
	DEFINT	DEFSNG	DEFDBL	\	MOD
AMSTRAD	DEFINT	—(1)	—(1)		MOD
APPLE II APPLESOFT	—	—	—	—	—
APRICOT (M-BASIC)	DEFINT	DEFSNG	DEFDBL	\	MOD
ATARI	—	—	—	—	—
CBM 64	—	—	—	—	—
DRAGON	—	—	—	—	—
EQUIPOS MSX	DEFINT	DEFSNG	DEFDBL	\	MOD
HP-150	DEFINT	DEFSNG	DEFDBL	\	MOD
IBM PC	DEFINT	DEFSNG	DEFDBL	\	MOD
MPF	—	—	—	—	—
NCR DM-V (MS-BASIC)	DEFINT	DEFSNG	DEFDBL	\	MOD
NEW BRAIN	—	—	—	—	—
ORIC	—	—	—	—	—
SHARP MZ-700 (MZ-BASIC)	—	—	—	—	—
SINCLAIR QL	—	—	—	DIV	MOD
SPECTRAVIDEO	DEFINT	DEFSNG	DEFDBL	\	MOD
ZX-SPECTRUM	—	—	—	—	—

OBSERVACIONES:
(1) DEFREAL declara variables reales implícitas

MOD

Calcula el valor del resto asociado a una división entera.

FORMATO: <dividendo> MOD <divisor>
EJEMPLOS:

```
20 LET TGN=17 MOD 3
70 PRINT A MOD 2
```

```
PRINT INT(5/4) <CR>
1
PRINT INT(4/5) <CR>
0
■
```

Sin embargo, la división entera guarda aún una sorpresa. Por medio de ella

es posible determinar también el resto de la división. Como quiera que el resto se denomina en inglés «module» (módulo), la función asociada a la anterior y que proporciona el valor del resto recibe el nombre de MOD:

```
PRINT 5%\4% <CR>
1
PRINT 5% MOD 4% <CR>
1
PRINT 4%\5% <CR>
0
PRINT 4% MOD 5% <CR>
4
```

Los ejemplos reflejados ilustran el uso combinado de ambos operadores: división entera y módulo de un cociente.



Del microprocesador al microordenador

En cualquier disciplina del saber humano, para conocer a fondo un sistema es necesario saber cómo está constituido.

Para poder manejarlo con soltura hace falta tener presente su organización.

Un microordenador es un sistema basado en microprocesador. Sin lugar a dudas, el conocimiento de su estructura interna permite entender mejor su funcionamiento y adquirir una idea clara de sus posibilidades.

Cualquier ordenador consta de un conjunto de bloques fundamentales. Estos son la unidad central de proceso, el reloj, la memoria y la unidad de entrada/salida. A estas unidades internas del ordenador es necesario añadir otros dispositivos externos: los denominados periféricos.

Cada una de estas zonas constitutivas del ordenador tiene su cometido específico. Todas ellas se comunican entre sí por medio de conductores de enlace que canalizan la información en forma de señales eléctricas. Estos cables pueden transmitir distintos tipos de información: datos y órdenes. Cada mazo o agrupación de conductores que permite el intercambio de información entre los elementos del sistema se denomina BUS. La función de cada unidad básica de un sistema microprocesador es la que se indica a continuación.

Unidad central de proceso (CPU): es el centro neurálgico del microordenador y está constituida por el circuito integrado microprocesador. Se encarga de realizar las operaciones aritméticas y lógicas y de interpretar las instrucciones del programa.

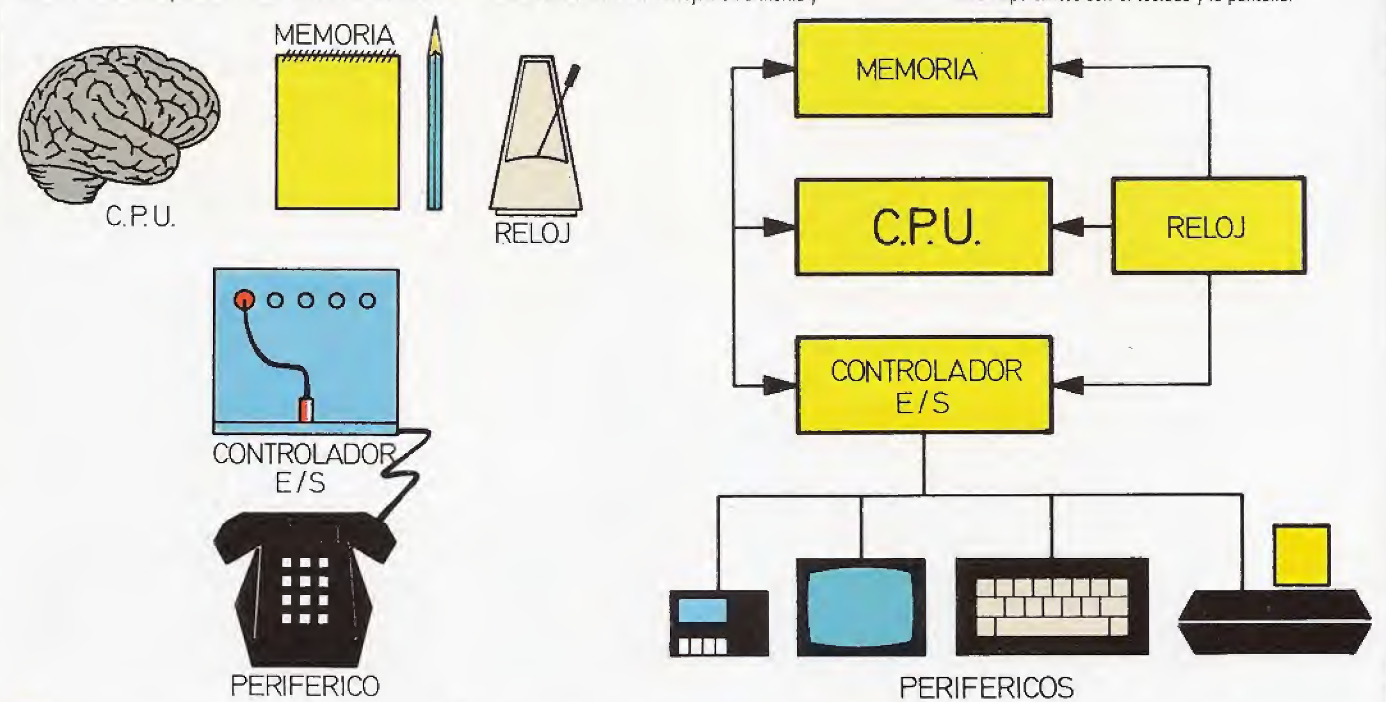
La CPU es el auténtico cerebro del sistema y controla el funcionamiento de los restantes elementos y unidades que lo constituyen.

Reloj: el reloj es al ordenador lo que el director a la orquesta. Es precisamente el que marca el ritmo de operación. Gracias a este circuito, el conjunto de elementos asociados trabajan en armonía y sincronizadamente. La frecuencia de reloj de un ordenador da una idea de la velocidad de operación del mismo.

Memoria: está integrada por una serie de dispositivos electrónicos capaces de almacenar información binaria. La memoria es el bloc de notas del ordenador. En ella anota los datos y resultados de sus cálculos, así como la lista de tareas a realizar (el programa). Por regla general, cuanto mayor sea la memoria más potente será el ordenador. Esta magnitud se mide en Kbytes (1Kbyte=1024 bytes u octetos de información).

Unidad de entrada/salida: su función es la de facilitar la comunicación del ordenador con los dispositivos externos. Se encargan de adecuar los formatos y velocidades de la información a intercambiar entre el ordenador y los dispositivos periféricos.

Periféricos: son los dispositivos físicos que permiten la comunicación con el exterior. Sin ellos el ordenador estaría aislado y sería incapaz de recibir y emitir información. En los microordenadores los dos periféricos más importantes son el teclado y la pantalla.



Los elementos básicos que dan cuerpo al ordenador tienen un gran paralelismo con otros asociados a la actividad humana: el cerebro o CPU (el microprocesador, en los microordenadores), la memoria, el reloj, que sincroniza el ritmo de trabajo, y el controlador de entrada/salida, que facilita el intercambio de información con el mundo exterior a través de dispositivos periféricos.

Variables suscritas

Conjuntos de variables de múltiples dimensiones



Para trabajar en BASIC con datos que no poseen un valor fijo, hay que recurrir a las variables. Como ya sabemos, su cometido no es otro que «guardar» o memorizar los datos que les sean asignados. Así, por ejemplo, en la variable MEDIA se puede almacenar la nota media por alumno de un determinado curso. Si se quiere diferenciar las notas medias de los distintos cursos habrá que utilizar más variables. Por ejemplo: MEDIA1, MEDIA2,... A partir de todas ellas, se puede calcular la nota media total de los cursos de EGB. La forma de hacerlo es sencilla y bastaría con una sola línea de programa:

```
50 LET MEDEGB=(MEDIA1+MEDIA2+
MEDIA3+MEDIA4+MEDIA5+MEDIA6+
MEDIA7+MEDIA8)/8
```

No cabe duda que esta forma de trabajar con conjuntos resulta tediosa. El mayor inconveniente radica en la necesidad de acceder a cada dato individualmente. Ello puede resolverse en parte haciendo uso de un nuevo tipo de variables: las variables de conjunto, o variables suscritas.

En el ejemplo anterior se utilizaban ocho variables distintas para almacenar datos de características similares. Sus nombres sólo se diferenciaban en el último carácter, lo cual denotaba una naturaleza común.

Las ocho variables pueden considerarse como un conjunto homogéneo de datos, al que convendría dar un nombre genérico. Los elementos del conjunto debieran ser accesibles por medio de dicho nombre, acompañado por un indicativo para precisar el elemento elegido. Estas son, precisamente, las características de una variable de conjunto o «array» en BASIC. En notación matemática, los conjuntos de datos (por ejemplo, coeficientes de un polinomio) se suelen expresar por medio de un nombre genérico y un subíndice indicativo de cada elemento. En BASIC, el método es similar: un nombre de variable y un índice encerrado entre paréntesis. El conjunto de variables del ejemplo inicial podría expresarse de la siguiente forma:

```
MEDIA(1),MEDIA(2) ... MEDIA(8)
```

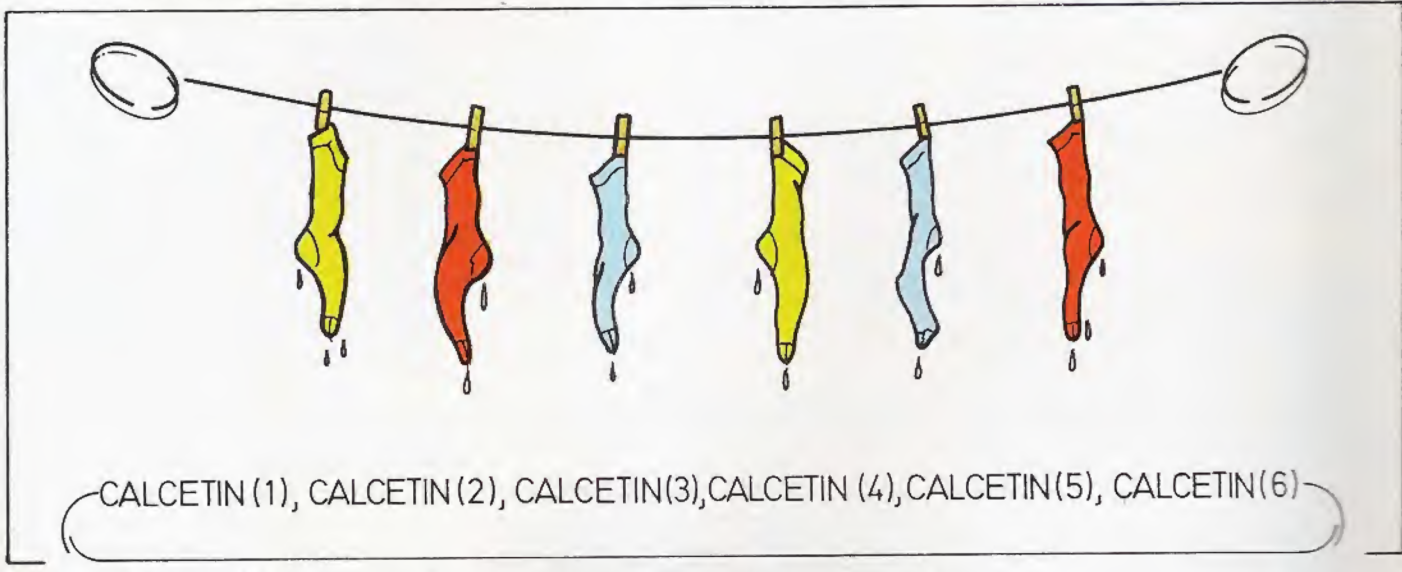
El nombre genérico del conjunto es MEDIA; éste va acompañado por el índice adecuado en cada caso. Tal como se observa, los índices son numéricos; más concretamente, números naturales. Este hecho proporciona algunas ventajas. En primer lugar, los índices definen un orden dentro del conjunto. Orden que puede reflejar una sucesión lógica de los datos (por ejemplo, la sucesión de las calificaciones académicas a lo largo de los sucesivos meses).

La segunda ventaja radica en la posibilidad de utilizar una segunda variable para almacenar el valor del índice. En efecto, los índices son datos numéricos, y como tales pueden ser tratados de forma matemática. El siguiente ejemplo refleja un método válido para acceder al cuarto elemento de un «array» o conjunto de variables suscritas:

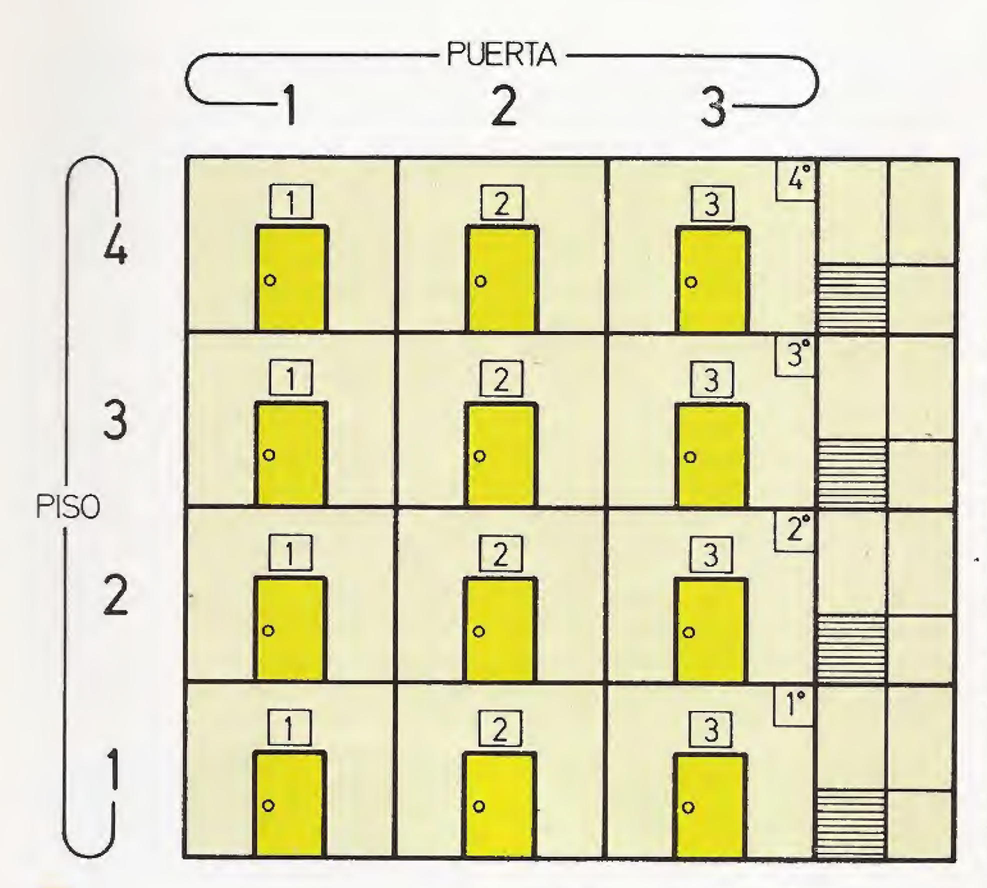
```
30 LET INDICE=4
40 PRINT MEDIA(INDICE)
```

Esta característica otorga una potencia y flexibilidad importantes a las variables de conjunto. Así, por ejemplo, las operaciones con «arrays» se pueden realizar de forma recursiva. Esto es: definiendo la operación para un elemento genérico y haciendo variar el índice para cubrir todos los elementos del conjunto. La presentación en la pantalla puede ordenarse, en consecuencia, con extremada sencillez:

```
120 FOR INDICE=1 TO 8
130 PRINT MEDIA(INDICE)
140 NEXT INDICE
```



Una variable suscrita, o «array», de una sola dimensión, puede representarse como una línea de elementos que obedecen a un mismo nombre de variable y que se identifican por su respectivo índice.



Cuando un conjunto de variables posee dos índices nos encontramos ante una matriz de dos dimensiones. Un edificio de varios pisos, con varias puertas por planta, constituye un ejemplo ilustrativo de esta circunstancia. Cada apartamento se identifica con dos índices: piso, puerta.

En el ejemplo se ha definido la operación a realizar una sola vez (línea 130). El bucle FOR/NEXT será el encargado de «barrer» todos los posibles valores del índice (del 1 al 8), con lo que la operación afectará sucesivamente a todos los elementos del «array».

Adoptando este método, el ejemplo para el cálculo de la media total de los cursos de EGB se simplifica en gran medida:

```
50 LET SUMA=0
60 FOR I=1 TO 8
70 LET SUMA=SUMA+MEDIA(I)
80 NEXT I
90 MEDEGB=SUMA/8
■
```

En este caso, la operación consiste en añadir el valor del elemento genérico a la variable SUMA. Al acumularlos todos en SUMA, se habrá obtenido la suma total de los ocho elementos. Para evaluar la calificación media basta tan sólo con dividir el valor de SUMA entre 8, al producirse la salida del bucle (línea 90). El cometido de la línea 50 no es otro que «borrar» el contenido que anteriormente pudiera tener la variable SUMA, para tener la seguridad de que sólo contribuirán a la suma total los elementos de MEDIA.

La segunda dimensión

Los conjuntos de variables utilizados hasta ahora se denominan «lineales» o de *una dimensión,* ya que sólo poseen un índice. En efecto, los elementos de estos «arrays» pueden ordenarse en una línea, al contar con un único índice de referencia.

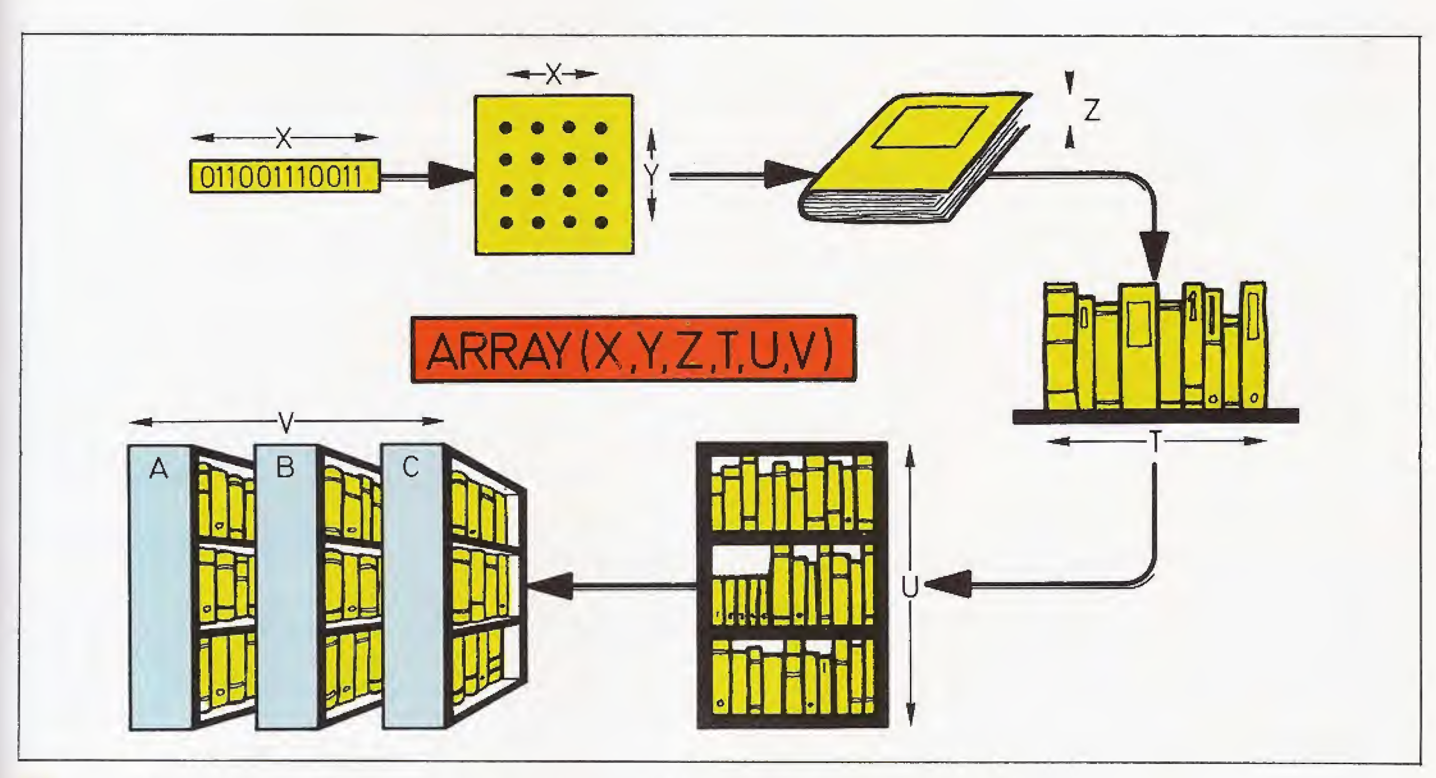
Estos conjuntos de variables resultan útiles para el tratamiento de datos con una sola característica diferenciadora (un solo «grado de libertad»). Así, por ejemplo, en el conjunto de notas medias, la característica identificativa era el curso de EGB. En determinadas ocasiones, es preciso diferenciar más de una cualidad de cada dato. Por ejemplo, el conjunto de notas medias de varios alumnos. Ahora, aparte del identificador de curso, habrá que introducir el identificador del alumno. Ya no basta con determinar el curso al que se hace referencia, puesto que cada alumno tendrá un dato distinto de nota media en dicho curso. Este segundo conjunto puede representarse como una tabla, en la que las filas corresponden a los alumnos y las columnas a los cursos. En definitiva, la ordenación se realiza en dos direcciones: se trata, pues, de un conjunto de *dos dimensiones.*

Para acceder a estos nuevos «arrays» o matrices, son necesarios dos índices. Su formulación en BASIC tiene el siguiente aspecto: MEDIA(3,7).

Los dos índices se encuentran encerrados entre paréntesis y separados entre sí por una coma. La forma de trabajo es análoga a la que corresponde a los conjuntos de variables de una sola dimensión. Por ejemplo, para visualizar las notas del alumno número uno, las instrucciones adecuadas serían:

```
120 FOR INDICE =1 TO 8
130 PRINT MEDIA(INDICE,1)
140 NEXT INDICE
■
```

Su similitud con el caso de una sola dimensión es obvia. En realidad, un conjunto de variables de dos dimensiones viene a ser un «conjunto de subconjuntos»; con el primer índice se numeran los elementos de cada subconjunto y con el segundo los subconjuntos.



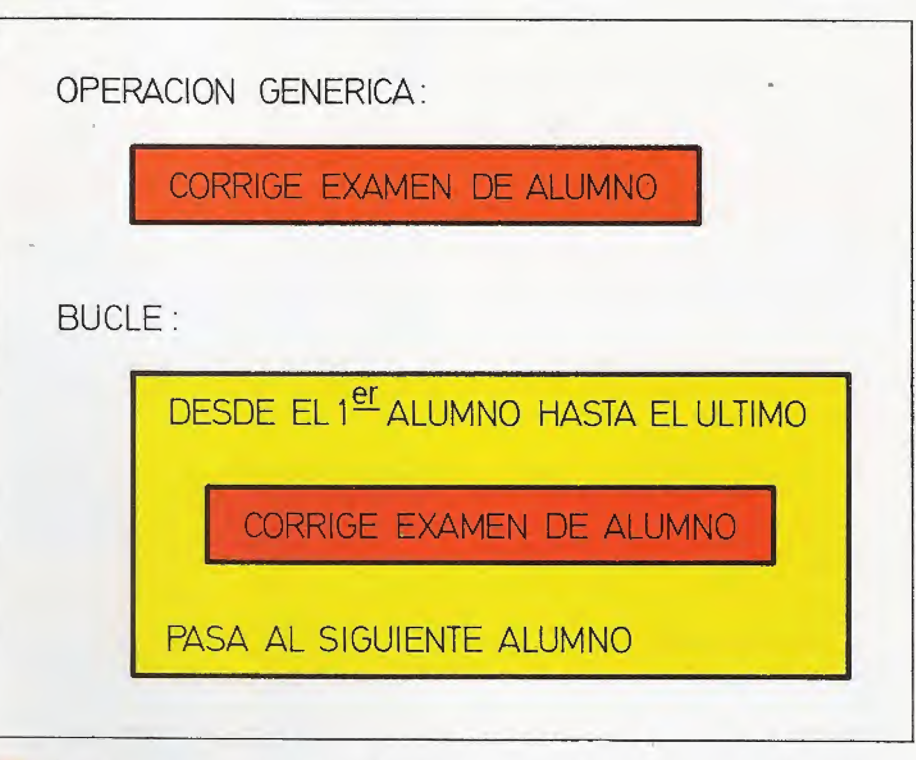
Los arrays o matrices son conjuntos de un número cualquiera de dimensiones. La figura aporta ejemplos sugerentes de conjuntos de variables de una a seis dimensiones.

Los conjuntos de dos dimensiones resultan perfectamente adecuados para almacenar matrices. En tal caso, uno de los índices representa a la fila y el otro a la columna que ocupa cada elemento. La elección del índice que corresponde a la fila o a la columna es totalmente libre, siempre que el resto de las operaciones a realizar sea consecuente con esta elección.

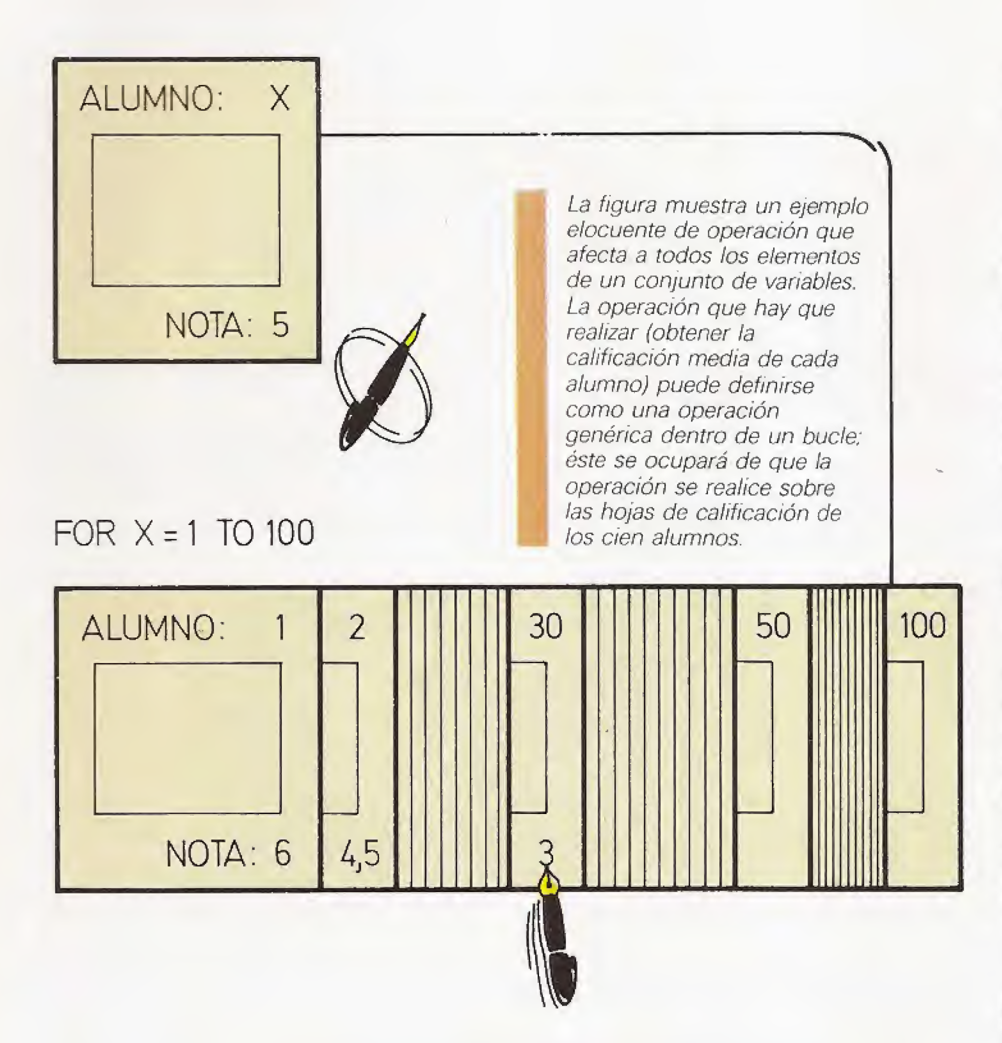
Para tratar recursivamente un «array» de dos dimensiones no basta con un bucle FOR/NEXT, sino que hay que construir un bucle por cada una de las dimensiones. Uno de ellos recorrerá los elementos de cada fila (columna a columna) y el otro irá recorriendo las sucesivas filas.

Pasemos al terreno práctico, suponiendo que el primer índice representa la fila y el segundo al elemento específico que ocupa una determinada posición dentro de la misma. La operación a realizar coincidirá con una asignación de valor a los elementos de la matriz. La siguiente línea de instrucción resulta apropiada para tal cometido:

```
40 LET MATRIZ(FILA,ELEMENTO)=0
```



Para ejecutar una operación con todos los elementos de un conjunto de variables es preciso definirla sobre un conjunto genérico y encerrarla dentro de un bucle repetitivo que afecte, sucesivamente, a todos los elementos.



La figura muestra un ejemplo elocuente de operación que afecta a todos los elementos de un conjunto de variables. La operación que hay que realizar (obtener la calificación media de cada alumno) puede definirse como una operación genérica dentro de un bucle; éste se ocupará de que la operación se realice sobre las hojas de calificación de los cien alumnos.

Desde luego, esta línea representa a una asignación genérica. Ahora, es necesario crear el bucle que recorra todos los elementos de una determinada fila. En el ejemplo, supondremos que cada fila consta de diez elementos:

```
30 FOR ELEMENTO=1 TO 10
40 LET MATRIZ(FILA,ELEMENTO)=0
50 NEXT ELEMENTO
```

Tres instrucciones BASIC son suficientes para asignar un valor (el cero en nuestro caso) a todos los elementos de una fila.

A continuación trataremos a estas tres líneas de programa como un bloque unitario, para extender la asignación a las seis filas de elementos que consideramos en nuestra matriz ejemplo. Para ello, crearemos un nuevo bucle externo que recorra todas las filas:

```
30 FOR FILA=1 TO 6
30   FOR ELEMENTO=1 TO 10
40     LET MATRIZ(FILA,ELEMENTO)=0
50   NEXT ELEMENTO
70 NEXT FILA
█
```

Ambos bucles rodean a la línea de instrucción genérica que se ocupa de ordenar la asignación (línea 40). Se observa que los dos bucles aparecen «anidados». Cabe recordar en este punto que, cuando se crean bucles anidados, es indispensable cerrar con la palabra clave NEXT los bucles interiores antes que los exteriores.

Conjuntos de múltiples dimensiones

Si un conjunto de una dimensión equivalía a puntos situados en línea, y uno de dos dimensiones a una tabla o retícula (algo semejante a un tablero de ajedrez), la tercera dimensión cabe imaginarla como un conjunto ordenado según las tres direcciones espaciales. En este caso, además de filas y columnas cabe pensar en una tercera referencia, por ejemplo, en el «nivel de profundidad». Los «arrays» de tres dimensiones se tratan igual que los de dos o una dimensión. La única diferencia radica en que ahora serán tres los índices y los bucles anidados para su tratamiento recursivo.

Volviendo al ejemplo inicial, si la primera dimensión indicaba las notas de un alumno y la segunda reflejaba a los distintos alumnos, cabe la posibilidad adicional de agrupar a los alumnos por colegios. En tal caso, el tercer índice será el indicativo de los diversos colegios considerados. Con esta estructura tridimensional, será posible acceder a la nota del curso X, concretamente del alumno Y, que cursa sus estudios en el centro Z.

Pero las dimensiones no acaban aquí. En BASIC se puede saltar a la cuarta dimensión, a la quinta, a la sexta,...

Por cada nueva dimensión la cosa se complica un poco más. Es necesario incluir un nuevo índice y, para realizar una operación con todos los elementos del conjunto, es preciso anidar un nuevo bucle FOR/NEXT. Afortunadamente, el trabajo habitual se reduce a «arrays» de a lo sumo tres o cuatro dimensiones.

Dando tamaño a los conjuntos

El tamaño de los conjuntos de variables con los que se puede trabajar en un ordenador no es ilimitado. Como es lógico, depende de la capacidad de memoria del equipo.

Cuando se define una variable, el ordenador se encarga de «reservarle sitio» automáticamente. Cuando se trata de conjuntos o «arrays» el proceso es más complejo, puesto que exigen un mayor

espacio de memoria. Además, el espacio necesario va a depender del número máximo de elementos del conjunto. Para evitar que en un momento dado no quede suficiente memoria para almacenar un «array», es necesario reservar previamente el espacio preciso.

Esta «reserva» de espacio se realiza en BASIC mediante una instrucción llamada DIM (de DIMensionar). En el argumento de DIM se especifica el nombre del «array», seguido por la longitud máxima de sus dimensiones. Para reservar el espacio necesario para la tabla de calificaciones de los 8 cursos de EGB, cada uno con 10 alumnos, habrá que utilizar la instrucción:

```
DIM MEDIA(8,10)
```

De esta forma el ordenador «toma conciencia» de la magnitud del conjunto de variables.

En términos generales, el formato de DIM es el siguiente:

(Número de línea) DIM <lista de variables del conjunto>

Habitualmente, se puede dimensionar más de un «array» en cada instrucción DIM. Esto se consigue separando los conjuntos a definir por medio de comas. Por ejemplo:

```
DIM GASTOS(12),INGRESOS(12),
TOTALES(3,12)
```

El dimensionamiento de los «arrays» o conjuntos de variables suscritas, suele realizarse al principio del programa. De esta forma, no cabe el riesgo de operar con un conjunto de variables que no haya sido definido previamente. En la mayoría de los ordenadores, existe un método de dimensionado automático

TABLA DE CONVERSION

Ordenador	DIM		OPTION BASE
	DIM <var.>	DIM <lista>	OPTION BASE <ind.>
AMSTRAD	DIM <var.>	DIM <lista>	—
APPLE II (APPLESOFT)	DIM <var.>	DIM <lista>	—
APRICOT	DIM <var.>	DIM <lista>	OPTION BASE <ind.>
ATARI	DIM <var.>	DIM <lista>	—
CBM 64	DIM <var.>	DIM <lista>	—
DRAGON	DIM <var.>	DIM <lista>	—
EQUIPOS MSX	DIM <var.>	DIM <lista>	—
HP-150	DIM <var.>	DIM <lista>	OPTION BASE <ind.>
IBM PC	DIM <var.>	DIM <lista>	OPTION BASE <ind.>
MPF	DIM <var.>	—	—
NCR DM-V (MS-BASIC)	DIM <var.>	DIM <lista>	OPTION BASE <ind.>
NEW BRAIN	DIM <var.>	DIM <lista>	OPTION BASE <ind.>
ORIC	DIM <var.>	—	—
SHARP MZ-700 (MZ-BASIC)	DIM <var.>	DIM <lista>	—
SINCLAIR QL	DIM <var.>	DIM <lista>	—
SPECTRAVIDEO	DIM <var.>	DIM <lista>	—
ZX-SPECTRUM	DIM <var.>	—	—

<var.>: Nombre de variable de conjunto o «array». <lista>: Lista de variables de conjunto. <ind.>: Indice inicial (0 ó 1).

FORMULACIONES DE LOS COMANDOS

DIM <var.>: Reserva en memoria para una variable de conjunto. DIM <lista>: Reserva espacio para varios «arrays» por medio de un solo comando DIM. OPTION BASE: Fija el índice inicial del «array».

DIM

Dimensiona el tamaño máximo de una o varias variables de conjunto.

Formato: (Número de línea) DIM <lista de variables de conjunto>

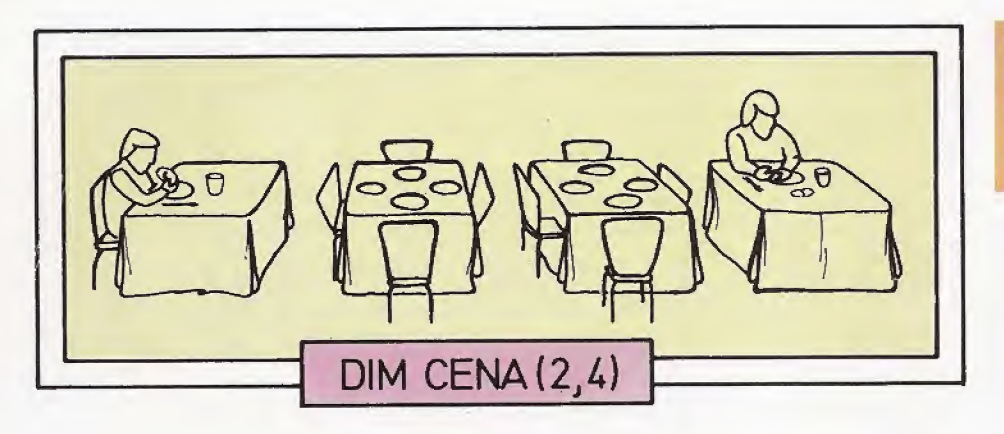
Ejemplos: 10 DIM A(100)
50 DIM OROS(10),COPAS(10),BAR(4,10)

OPTION BASE

Permite elegir el valor mínimo del índice entre uno y cero.

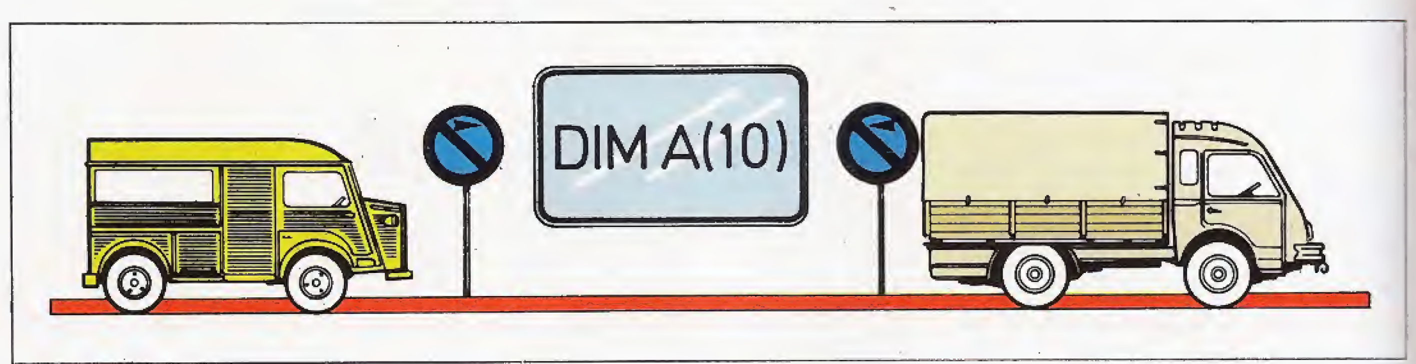
Formato: (Número de línea) OPTION BASE *<índice>*

Ejemplos: 10 OPTION BASE 0
12 OPTION BASE 1



Al formular una instrucción DIM, hay que especificar los valores que pueden adoptar los índices de la variable suscrita. En el ejemplo de la figura, la instrucción DIM reserva dos mesas de cuatro comensales cada una.

La función del comando DIM es reservar espacio en memoria para almacenar todos los elementos que constituyen el conjunto de variables o matriz.



Ejemplo de «array» de dos dimensiones

El siguiente programa ilustra el tratamiento de una variable suscrita de dos dimensiones. Su cometido se reduce a definir una matriz de 3×3 elementos (línea 5), solicitar la introducción de los nueve datos y, finalmente, mostrarlos en la pantalla. Tal y como se ha descrito en el texto, la manipulación recursiva de un conjunto de variables de doble dimensión (dos índices), exige la presencia de dos bucles anidados. En el caso que nos ocupa, los valores iniciales de las instrucciones FOR especifican el valor 1 como primer índice.

```
  5 DIM A(3,3)
 10 FOR I=1 TO 3
 20 FOR J=1 TO 3
 30 PRINT "INTRODUCE EL ELEMENTO
    A(";I;",";J;")"
 40 INPUT A(I,J)
 50 NEXT J
 60 NEXT I
100 FOR I=1 TO 3
110 FOR J=1 TO 3
120 PRINT A(I,J),
130 NEXT J
140 PRINT
150 NEXT I
160 END
```

La ejecución del programa se traduce en la siguiente secuencia en la pantalla del ordenador:

```
INTRODUCE EL ELEMENTO A(1,1)
1
INTRODUCE EL ELEMENTO A(1,2)
2
INTRODUCE EL ELEMENTO A(1,3)
3
INTRODUCE EL ELEMENTO A(2,1)
4
INTRODUCE EL ELEMENTO A(2,2)
5
INTRODUCE EL ELEMENTO A(2,3)
6
INTRODUCE EL ELEMENTO A(3,1)
7
INTRODUCE EL ELEMENTO A(3,2)
8
INTRODUCE EL ELEMENTO A(3,3)
9

1 2 3
4 5 6
7 8 9
```

para «arrays» pequeños. Si no se fija la longitud, el traductor BASIC supone que se trata de un conjunto de 10 elementos. Así pues, la línea de instrucción 10 DIM A(10), B(10), C(10) puede omitirse si el ordenador posee esta característica.

El elemento cero

Hemos comentado que los índices definen una ordenación: desde el 1 hasta el valor máximo que tomen. En algunos casos —sobre todo en aplicaciones matemáticas—, resulta interesante disponer del índice cero. Al respecto cabe señalar que algunos dialectos BASIC no admiten la presencia del índice cero, mientras que otros dialectos lo incorporan a cualquier «array» definido. Un tercer grupo de traductores BASIC permite elegir la opción adecuada.

Los ordenadores que permiten elegir el índice del primer elemento poseen, al efecto, el comando OPTION BASE. En el argumento de OPTION BASE se especifica cuál debe ser el índice inferior: 1 ó 0.

Aportando datos a la máquina

Nuevos comandos para el suministro de información



Cualquier programa, sea cual fuere su naturaleza (de gestión, educativo o, sencillamente, lúdico) necesita manipular una cierta cantidad de datos, que deben ser entregados a la máquina para que los elabore. Estos pueden incluirse dentro del propio programa, o incluso puede suministrarlos el usuario, en un determinado momento, a través del teclado.

Diferenciación de los datos

En términos generales cabe hablar de dos tipos de datos o, más exactamente, de dos formas de aportarlos al programa: como valores constantes o a modo de variables.

Aún cabe establecer entre los datos otra división, quizá no tan clara y definida como la anterior, pero no por ello menos válida. Esta división nace de su propia utilidad: ciertos datos van a mantener valores fijos y determinados en las sucesivas ejecuciones del programa; por el contrario, otros verán alterado su valor, ya que el programa los modificará para ofrecer al usuario el resultado de la ejecución.

Un ejemplo clarificará este extremo. Si se quiere obtener al cabo de un año el total de gastos mensuales de una familia, habrá que utilizar al menos dos variables con distinta función. Por un lado, el programador debe tener presente que el año consta de doce meses y, por lo tanto, habrá que considerar doce entradas de datos, una para cada mes del año. Por otro lado, hay que contar con otra variable que irá acumulando las sucesivas cantidades mensuales.

La actividad del ordenador supone la manipulación de datos, la ejecución de cálculos y, en definitiva, el tratamiento de los mismos de acuerdo a la secuencia de órdenes denominada programa.



Un programa capaz de realizar estos cálculos es, por ejemplo, el que sigue:

```
10 LET SUMA=0
20 FOR M=1 TO 12
30 INPUT A
40 LET SUMA=SUMA+A
50 NEXT M
60 PRINT "GASTO ANUAL="; SUMA
70 END
█
```

Su ejecución interrogará al usuario acerca de los sucesivos gastos mensuales, hasta completar la introducción de datos mensuales; tras ello, mostrará el valor del gasto anual.

```
RUN
?35650
?42500
...
?15250
GASTO ANUAL=525345
```

El programa propuesto comienza por inicializar a cero la variable SUMA para acumular en ella los doce valores, ingresados a través del teclado y asignados a la variable A; ésta forma parte de un bucle que se ejecutará doce veces.

Las dos variables en juego, SUMA y A, adoptarán valores muy diferentes en cada ejecución del programa: los gastos de un mes serán, por lo general, muy distintos a los de otro. Sin embargo, la variable de control del bucle FOR/NEXT (M), siempre adoptará los valores comprendidos entre uno y doce.

Desde luego que éste es un ejemplo trivial; en todo caso, existen muchas variables dentro de un programa que tienen el carácter fijo de la variable M. Por ejemplo: los nombres de los meses, el nombre y la fecha de nacimiento de los miembros de una familia o de los trabajadores de una empresa, los resultados de los partidos de fútbol del año anterior... A este tipo de variables de carácter fijo, se les podría asignar, como es lógico, sus valores mediante instrucciones INPUT. Valores que se introducirán a través del teclado cada vez que se eje-



El lenguaje BASIC dispone de un nutrido grupo de comandos para la introducción de datos, una de las funciones primordiales de cualquier programa.

cute el programa. No obstante, y dado que estos valores son siempre los mismos, es obvio pensar que se ahorraría mucho tiempo y molestias, si su asignación corriera a cargo del propio programa.

Una forma elemental de almacenar los referidos datos por medio del programa la aporta la instrucción LET, adecuada para asignar directamente los valores deseados a las variables. Así, por ejemplo, se podrían almacenar los nombres de los días de la semana de la siguiente forma:

```
10 LET D1$="LUNES"
20 LET D2$="MARTES"
30 LET D3$="MIERCOLES"
40 LET D4$="JUEVES"
50 LET D5$="VIERNES"
60 LET D6$="SABADO"
70 LET D7$="DOMINGO"
█
```

Ello hará que el programa los tenga a su disposición en cualquier momento, obviando la necesidad de introducirlos cada vez que se desee ejecutarlo. Si el número de variables fuese reducido, ésta sería una opción viable y bastante eficaz, pero ¿qué ocurriría si el número de variables de este tipo fuese elevado?

Leyendo los datos

Naturalmente, si hubiera que almacenar, por ejemplo, los resultados de los partidos de fútbol de la temporada pasada a base de instrucciones LET o INPUT, sería una tarea larga y tediosa. Afortunadamente, el BASIC cuenta con otras instrucciones que resuelven este problema con mayor comodidad.

Las referidas instrucciones son READ y DATA. Ambas deben coexistir obligatoriamente dentro de un programa, esto es: si se utiliza una instrucción READ, debe existir forzosamente al menos una instrucción DATA. La razón se comprenderá fácilmente una vez que se conozcan las funciones que realizan cada una de ellas y que, en ensencia, se resumen en lo siguiente: las instrucciones DATA contienen los valores fijos que se desea asignar posteriormente a las variables, mientras que las instrucciones READ se encargan de leer el argumento de la instrucción DATA y asignar los valores de su argumento a las variables especificadas en la instrucción READ.

Formatos de READ y DATA

Como toda instrucción BASIC, tanto READ como DATA deben ajustarse a un formato sintáctico que es preciso respetar escrupulosamente. De lo contrario, la instrucción no será aceptada por el intérprete BASIC y la máquina generará un mensaje «error sintáctico».



El cometido del ordenador es manipular datos de acuerdo a las indicaciones aportadas por el programa. Los datos pueden suministrarse a través de dos vías principales: el teclado, y formando parte del propio programa.

El formato de la instrucción READ es el siguiente:

(Núm. línea) READ <var.>], <var.>, ..., <var.>]

La palabra clave READ debe ir precedida por el siempre obligatorio número de línea, propio de las instrucciones formuladas en modo indirecto (dentro de un programa). su argumento incluirá a una serie de variables, separadas por comas, cuyo número es opcional; aunque siempre debe estar presente al menos una variable. Por ejemplo:

```
10 READ A
20 READ A, B, C$
```

A su vez, el formato de una instrucción DATA es el que sigue:

(Núm. línea) DATA <cte.>[,<cte.>, ..., <cte.>]

Este formato es muy similar al anterior. Ahora, la palabra clave, DATA, irá seguida por al menos un dato; éste puede ser una constante de tipo numérico o alfanumérico (cadena de caracteres). Normalmente serán varios los datos incluidos en el argumento y que aparecerán separados por comas:

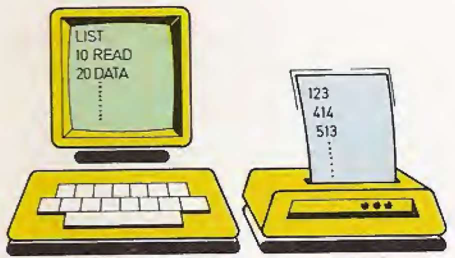
```
10 DATA 1,2,5,20,3
20 DATA LUNES
```

Las instrucciones DATA son leídas por las instrucciones READ, de acuerdo al orden que establecen sus números de línea aunque éstos no sean correlativos. En muchos dialectos BASIC los datos alfanuméricos pueden escribirse sin encerrarlos entre comillas; por supuesto, excepto en el caso de que el literal contenga alguna coma (,), con objeto de que la máquina no la confunda con las comas utilizadas para separar a los distintos datos. Esta observación es extensiva a las cadenas de caracteres que incluyan el signo dos puntos (:), para no confundirlo con el separador de dos instrucciones en la misma línea, o espacios en blanco significativos detrás o delante de la constante. Por ejemplo:

```
DATA MARTIN, "SANCHEZ," ,ENRIQUE
DATA "HORA:", "MINUTOS"
```

Las instrucciones DATA incluirán valores de tipo numérico o literal. No se admiten expresiones numéricas, ya sea con operaciones matemáticas o lógicas entre los datos.

Así, por ejemplo, no serán aceptadas las siguientes instrucciones:



En términos generales, los datos pueden incluirse dentro de los programas adoptando la forma de variables o de valores constantes. Su tratamiento dará lugar a la generación del resultado o resultados del proceso.

```
10 DATA 3/4
20 DATA A AND B,C OR B
```

En ambos ejemplos, los valores aportados en las sentencias DATA podrían ser leídos como cadenas de caracteres, pero no resultarán válidos si su misión es aportar el valor resultante de las expresiones aritméticas o lógicas.

READ y DATA: una pareja indisoluble

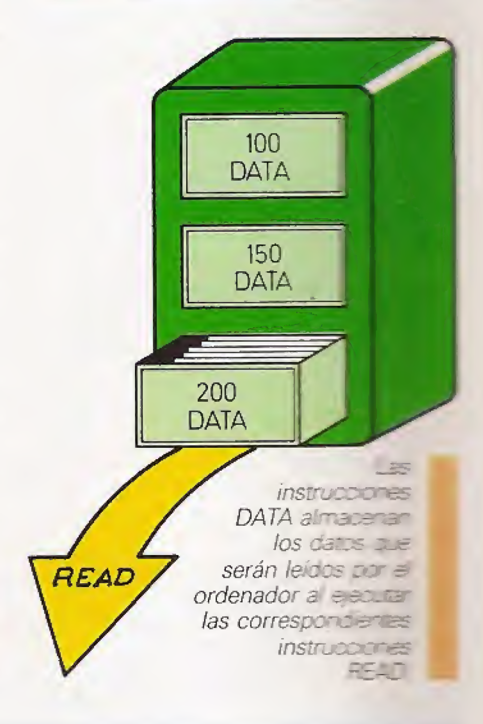
Una vez presentado el formato de ambas instrucciones, cabe analizar su funcionamiento y la forma de utilizarlas dentro de un programa.

Tal como se mencionó anteriormente, si en un programa se utiliza una instrucción READ para la lectura de datos, debe estar presente en el mismo programa al menos una instrucción DATA. Ello resulta imprescindible para que READ pueda leer valores y asignarlos a las variables especificadas en su argumento.

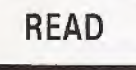
La instrucción READ asignará los valores que lea en la sentencia DATA de la siguiente forma: a la primera variable contenida en READ le asignará el valor de la primera constante que aparezca en la primera instrucción DATA (atendiendo al orden de numeración de las líneas); a la segunda variable, le asignará el segundo valor constante que lea en el correspondiente DATA, y así sucesivamente hasta que quede con asignación la última variable incluida en la instrucción READ.

Veamos un ejemplo:

```
10 READ A,B,C
20 PRINT A+B+C
30 DATA 15,30,10
40 END
█
```



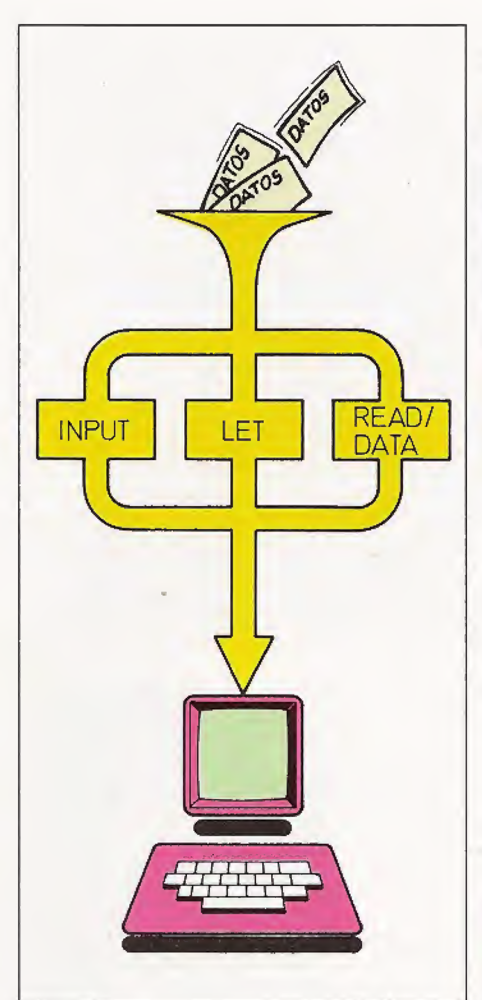
Las instrucciones DATA almacenan los datos que serán leídos por el ordenador al ejecutar las correspondientes instrucciones READ.



Lee datos de una instrucción DATA y los asigna a las variables especificadas en su argumento.

Formato: (NI) READ <var.> [,<var.>, ..., <var.>]

Ejemplos:
```
10 READ A
20 READ A, B, C
40 READ Z$, DOT$, A
```



Tres son las herramientas fundamentales que proporciona el BASIC para suministrar datos a la máquina: las sentencias INPUT, LET, y la asociación READ/DATA.

En este caso, la instrucción READ asignará a la variable A el valor 15, a la variable B el valor 30 y a la variable C el valor 10.

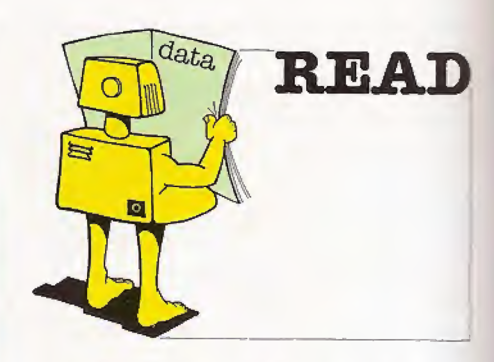
La ejecución del programa mostrará en la pantalla el siguiente resultado:

```
RUN
55
█
```

Los valores a asignar a las variables que acompañan a READ no tienen por qué estar contenidos en una única instrucción DATA, sino que pueden estar distribuidos en varias. La instrucción READ leerá los valores de la primera DATA hasta que agote sus datos; a continuación, empezará a leer de la siguiente instrucción DATA, prosiguiendo con una tercera sentencia DATA si fuera necesario, hasta completar la asignación. Así, el programa que sigue conduce al mismo resultado que el propuesto en el ejemplo anterior:

```
10 READ A,B,C
20 PRINT A+B+C
30 DATA 15
40 DATA 30
50 DATA 10
60 END
█
```

Algo análogo ocurre con las instrucciones READ, ya que su formato es se-

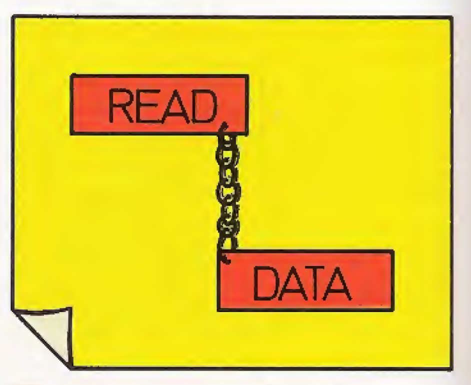


mejante al de las sentencias DATA. En consecuencia, una tercera alternativa para el mismo ejemplo será:

```
10 READ A
20 READ B
30 READ C
40 PRINT A+B+C
50 DATA 15
60 DATA 30
70 DATA 10
80 END
█
```

Si el número de variables a asignar con READ, o el número de constantes que hay que aportar, son considerables, será preciso utilizar varias sentencias READ y varias sentencias DATA para cumplimentar las asignaciones.

Las instrucciones READ y DATA pueden estar distribuidas en cualquier zona del programa; si bien, hay que respetar el orden en el que deben ser leídas. Una



El funcionamiento del par READ/DATA es indisoluble. Cualquier instrucción READ exige la presencia de una instrucción DATA que aporte los valores necesarios.

DATA

Almacena datos numéricos o cadenas de caracteres para su lectura por medio de instrucciones READ.

Formato: (Nl) DATA <const.> [, <const.>, ..., <const.>]

Ejemplos:
```
10 DATA 1,5,7,30.3,0
60 DATA 150, PTAS
85 DATA LUNES, 20, ABRIL
```

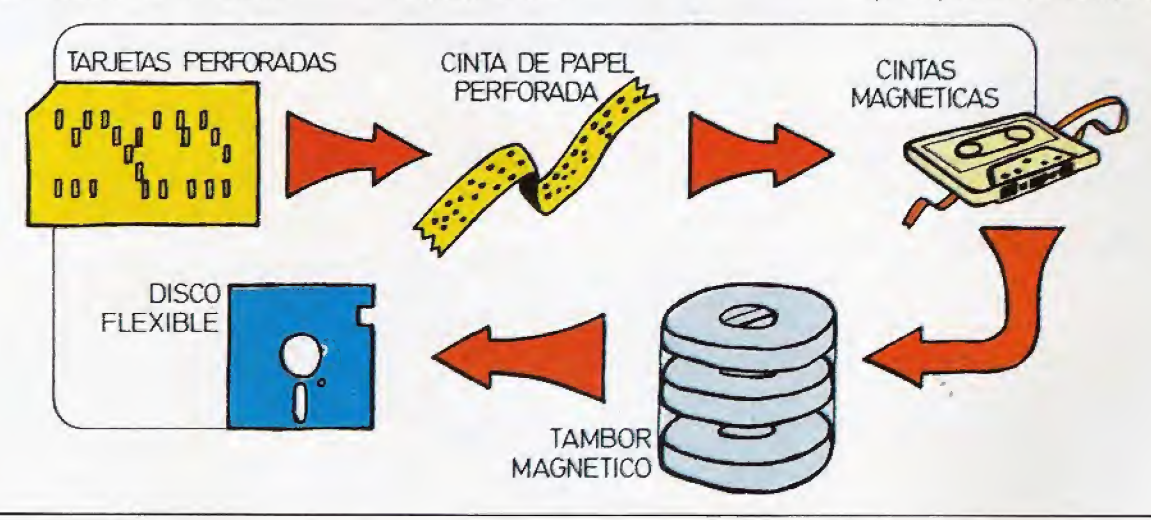
costumbre generalizada es agrupar a todas las instrucciones DATA para dar mayor claridad al listado del programa, aunque ello no debe considerarse como una regla inquebrantable.

La naturaleza de las variables (numéricas o de cadena de caracteres) a las que se asignará un valor por medio de READ, debe coincidir con la naturaleza (numérica o alfanumérica) de las respectivas constantes incluidas en la correspondiente instrucción DATA. De lo contrario, se producirá un mensaje de «error de asignación» («type mismatch error», o algo similar), ya que no es admisible el intento de asignar una cadena de caracteres a una variable numérica o viceversa.



Evolución de las unidades de almacenamiento externo

Las memorias de masa o auxiliares son dispositivos periféricos destinados a almacenar de forma permanenete grandes volúmenes de información, ya sean programas o datos.
Un programa almacenado en una unidad de este tipo no es directamente ejecutable, sino que, para que ello sea posible, es preciso trasladarlo previamente a la memoria central del ordenador.
Esta categoría de dispositivos presenta una característica fundamental: la información almacenada no es volátil, o lo que es lo mismo, no se borra al desconectar la alimentación. Ello permite conservar los programas y datos para emplearlos en cualquier otra ocasión. Su actuación se fundamenta en el aprovechamiento de las propiedades magnéticas, o en el control de la variación de la estructura física, de determinados materiales que constituirán el soporte de almacenamiento.
Probablemente, el primer soporte empleado para almacenar datos y programas en un ordenador fue la tarjeta perforada. Estas empezaron a utilizarse en la industria textil tras su invención por Josep Jacquad. Más tarde, las tarjetas perforadas fueron empleadas por Hermann Hollerith en una máquina destinada a procesar la información referente al censo de los Estados Unidos.
Una tarjeta perforada consiste en una simple cartulina rectangular en la que se disponen 12 filas con 80 columnas cada una de posibles perforaciones. La presencia o ausencia de perforación en los diversos puntos es, precisamente, lo que permite representar distintas informaciones. Para trasladar la información a la tarjeta se emplea comúnmente un código especial, llamado código Hollerith; éste permite codificar cualquier carácter en una columna de la tarjeta.
La perforación de las tarjetas se realiza mediante unas máquinas especiales llamadas perforadoras de tarjetas. Su lectura puede llevarse a cabo por procedimientos mecánicos (escobillas) u ópticos (células fotoeléctricas). En la actualidad, este soporte ha quedado totalmente obsoleto y su presencia se reduce a instalaciones anticuadas de grandes ordenadores.
Las cintas de papel perforado constituyen otro de los soportes de almacenamiento en desuso en nuestros días. Son simples cintas de papel, en las que se practican las perforaciones adecuadas para codificar la información. Hoy en día están presentes en algunos teletipos.
Las propiedades magnéticas de determinados materiales empezaron a aprovecharse para construir los primeros soportes de memoria en cinta magnética.
En esencia, la constitución y funcionamiento de las unidades apropiadas no difiere de la de los magnetófonos a casete convencionales.
Tampoco el soporte difiere en exceso. Se trata de una cinta plástica sobre la que se deposita una capa de material magnetizable. Esta, que constituye la superficie exterior de la cinta, es la que memoriza la información en forma de dominios magnéticos en distinta orientación. Su tamaño, longitud, formato y capacidad varían según el modelo.
Las cintas magnéticas son soportes de tipo secuencial. Ello supone un inconveniente, puesto que para acceder a una información específica es necesario pasar por todas las informaciones que la preceden, con la consiguiente pérdida en tiempo de acceso. Semejante problema no es exclusivo de las cintas magnéticas, sino que afecta también a las tarjetas perforadas y a la cinta de papel.
Otro método, también basado en las propiedades magnéticas de algunos materiales, es el de los tambores magnéticos. Consisten en unos cilindros sobre los que se deposita una capa de material magnetizable capaz de retener la información. Esta se graba y se lee mediante un cabezal cuyo brazo se mueve en la dirección del eje de giro del tambor; realizando los giros adecuados del tambor, será posible actuar libremente sobre cualquier punto de la superficie del cilindro. Por consiguiente, el acceso a la información es directo y no secuencial.
En la actualidad, el sistema más extendido lo constituyen los discos magnéticos, ya sean de tipo flexible (floppy disc) o rígidos, además de ciertas variantes menos extendidas. En esencia, se trata de un disco de plástico sobre el que se realiza la consabida operación de depositar una capa de material magnético. Estos discos presentan la ventaja de permitir un acceso directo o aleatorio a cualquier punto de su superficie, ventaja que comparten con el tambor magnético.
Hoy en día se sigue investigando con empeño en este campo, y no sólo para desarrollar nuevos periféricos de almacenamiento —ahí están, por ejemplo, las unidades de disco óptico a láser—, sino también para mejorar la capacidad y eficacia de los discos y cintas magnéticas.



```
10 READ A, N$
20 DATA PEPE,35
30 PRINT N$; "TIENE"; A; "discos"
40 END
```

La ejecución de las líneas anteriores, producirá en la pantalla el mensaje de error señalado. Para solventarlo, habrá que modificar la línea 10 como sigue:

```
10 DATA 35,PEPE
```

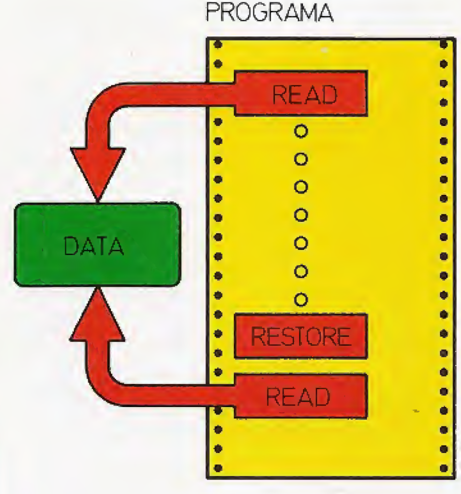
Ahora, la ejecución se realizará sin problema alguno:



Una de las ventajas de las instrucciones DATA radica en que para alterar los valores de los datos del programa, sólo hay que modificar esas líneas; no hay necesidad de cambiar cualquier otra instrucción operativa.

Cuando el número de constantes incluidas en una sentencia DATA no es fijo, sino que en una futura ejecución del programa puede verse ampliado con nuevos datos, es posible recurrir a un sencillo truco. Consiste en introducir un dato especial (distinto de cualquiera de los valores que lo preceden) al final de las constantes de la última sentencia DATA. Este servirá como indicador de que ya se han leído todos los datos anteriores. De esta forma, será posible modificar el número de constantes a leer sin alterar por ello el programa principal. El siguiente ejemplo ilustra la técnica enunciada:

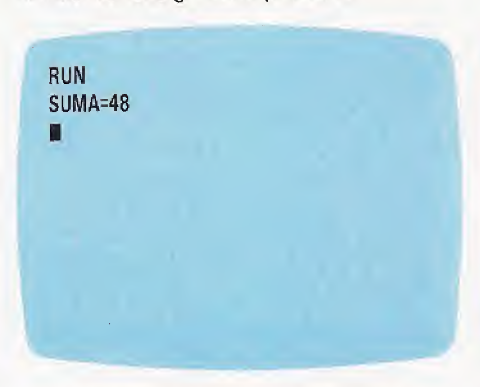
```
10 LET S=0
20 FOR C=0 TO 1 STEP 0
30 READ D
40 IF D=—1 THEN LET C=1
50 IF D< >—1 THEN LET S=S+D
60 NEXT C
70 PRINT "SUMA="; S
```



La instrucción RESTORE permite que el argumento de una misma instrucción DATA sea leído y, en consecuencia, utilizado, por sucesivas instrucciones READ.

```
80 DATA 10,2,8,23,5,—1
90 END
```

Su ejecución se traducirá en el contenido de la siguiente pantalla:



La instrucción READ irá leyendo los datos de la sentencia DATA, uno por uno, y acumulándolos en la variable S, hasta que llegue al valor −1, el último. Su lectura hará que la variable C que controla al bucle tome el valor 1, con lo que se abandonará el bucle FOR/NEXT y se imprimirá en la pantalla el resultado de la suma de los números aportados en la línea 80.

Si se desea modificar los datos a sumar, o simplemente añadir más números a la lista, será suficiente con alterar la línea 80, sin tocar para nada el resto del programa.

Así, por ejemplo, si se introduce la línea:

```
80 DATA 10,2,8,23,5,6,4,10,—1
```

en el lugar de la línea 80 original, la nueva ejecución del programa, obtendrá el siguiente resultado:



El comando READ se puede introducir también de forma directa. No ocurre así con el DATA que, generalmente, debe ser ejecutado de forma indirecta, dentro de un programa.

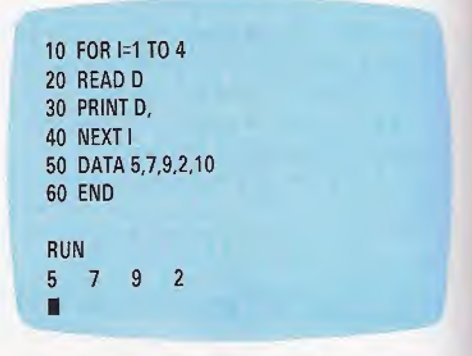




TABLA DE CONVERSION

Ordenador	READ		DATA		RESTORE		
	READ <var.>	READ <var.>,...,<var.>	DATA <const.>	DATA <const.>,...,<const.>	RESTORE	RESTORE <nl>	RESTORE <exp.>
AMSTRAD	READ <var.>	READ <var.>,...,<var.>	DATA <const.>	DATA <const.>,...,<const.>	RESTORE	RESTORE <nl>	—
APPLE II (APPLESOFT)	READ <var.>	READ <var.>,...,<var.>	DATA <const.>	DATA <const.>,...,<const.>	RESTORE	?	?
APRICOT (M-BASIC)	READ <var.>	READ <var.>,...,<var.>	DATA <const.>	DATA <const.>,...,<const.>	RESTORE	RESTORE <nl>	RESTORE <exp.>
ATARI	READ <var.>	READ <var.>,...,<var.>	DATA <const.>	DATA <const.>,...,<const.>	RESTORE	RESTORE <nl>	RESTORE <exp.>
CBM 64	READ <var.>	READ <var.>,...,<var.>	DATA <const.>	DATA <const.>,...,<const.>	RESTORE	—	—
DRAGON	READ <var.>	READ <var.>,...,<var.>	DATA <const.>	DATA <const.>,...,<const.>	RESTORE	—	—
EQUIPOS MSX	READ <var.>	READ <var.>,...,<var.>	DATA <const.>	DATA <const.>,...,<const.>	RESTORE	RESTORE <nl>	RESTORE <exp.>
HP-150	READ <var.>	READ <var.>,...,<var.>	DATA <const.>	DATA <const.>,...,<const.>	RESTORE	RESTORE <nl>	RESTORE <exp.>
IBM PC	READ <var.>	READ <var.>,...,<var.>	DATA <const.>	DATA <const.>,...,<const.>	RESTORE	RESTORE <nl>	RESTORE <exp.>
MPF	READ <var.>	READ <var.>,...,<var.>	DATA <const.>	DATA <const.>,...,<const.>	RESTORE	—	—
NCR DM-V (MS-BASIC)	READ <var.>	READ <var.>,...,<var.>	DATA <const.>	DATA <const.>,...,<const.>	RESTORE	RESTORE <nl>	RESTORE <exp.>
NEW BRAIN	READ <var.>	READ <var.>,...,<var.>	DATA <const.>	DATA <const.>,...,<const.>	RESTORE	RESTORE <nl>	RESTORE <exp.>
ORIC	READ <var.>	READ <var.>,...,<var.>	DATA <const.>	DATA <const.>,...,<const.>	RESTORE	—	—
SHARP MZ-700 (MZ-BASIC)	READ <var.>	READ <var.>,...,<var.>	DATA <const.>	DATA <const.>,...,<const.>	RESTORE	RESTORE <nl>	RESTORE <exp.>
SINCLAIR QL	READ <var.>	READ <var.>,...,<var.>	DATA <const.>	DATA <const.>,...,<const.>	RESTORE	RESTORE <nl>	RESTORE <exp.>
SPECTRAVIDEO	READ <var.>	READ <var.>,...,<var.>	DATA <const.>	DATA <const.>,...,<const.>	RESTORE	RESTORE <nl>	RESTORE <exp.>
ZX-SPECTRUM	READ <var.>	READ <var.>,...,<var.>	DATA <const.>	DATA <const.>,...,<const.>	RESTORE	RESTORE <nl>	RESTORE <exp.>

<nl>: Número de línea. <var.>: Variable. <const.>: Constante. <exp.>: Expresión numérica con datos y/o variables.

FORMULACIONES DE LOS COMANDOS

READ <var.>: Lee un dato de una instrucción DATA y lo asigna a la variable <var.>. READ <var.>, ..., <var.>: Lee tantos datos como variables haya y los asigna a las respectivas variables. DATA <const.>: Aporta un dato que será leído por medio de una instrucción READ, DATA <const.>, ..., <const.>: Aporta un conjunto de datos que serán leídos por una o varias instrucciones READ. RESTORE: Inicializa el puntero al primer dato de la primera instrucción DATA del programa. RESTORE <nl>: Inicializa el puntero de lectura al primer dato del DATA que ocupa la línea indicada. RESTORE <exp.>: Inicializa el puntero al DATA que ocupa la línea cuyo número resulta al evaluar la expresión.

RESTORE

Permite que el argumento de instrucciones DATA sea leído de nuevo por medio de instrucciones READ, desde el primer DATA o desde la línea DATA que se especifique.

Formato: (Nl) RESTORE [<nl>]
Ejemplos: 10 RESTORE
80 RESTORE 30

Si se introduce ahora la orden READ D y, acto seguido, se ejecuta la instrucción PRINT D, aparecerá en la pantalla el número 10. Sin embargo, al ejecutar una vez más la orden READ D, aparecerá en la pantalla un mensaje: «OUT OF DATA ERROR» (error de falta de datos).

Ello se debe a que el ordenador contabiliza interiormente el número de constantes DATA ya utilizadas, y al ter-

```
10 REM CALENDARIO PERPETUO (1980-1986)
20 PRINT "FECHA? (EJ.: 27,11,1984)"
30 INPUT D,M,A
40 IF D<1 OR D>31 OR M<1 OR M>12 THEN GOTO 20
50 IF A<1980 OR A>1986 THEN GOTO 20
60 RESTORE (200+A-1980)
70 FOR I=1 TO M
80 READ NUM
90 NEXT I
100 LET NUM=NUM+D
110 IF NUM>7 THEN LET NUM=NUM=7
120 IF NUM>7 THEN GOTO 110
130 RESTORE:LET A$=""
140 FOR I=1 TO NUM
150 READ A$
160 NEXT I
170 DATA DOMINGO,LUNES,MARTES
180 DATA MIERCOLES,JUEVES,VIERNES,SABADO
190 PRINT "EL ";D;" DEL ";M;" DE ";A;" ES ";A$
200 DATA 2,5,6,2,4,0,2,5,1,3,6,1
201 DATA 4,0,0,3,5,1,3,6,2,4,0,2
202 DATA 5,1,1,4,6,2,4,0,3,5,1,3
203 DATA 6,2,2,5,0,3,5,1,4,6,2,4
204 DATA 0,3,4,0,2,5,0,3,6,1,4,6
205 DATA 2,5,5,1,3,6,1,4,0,2,5,0
206 DATA 3,6,6,2,4,0,2,5,1,3,6,1
210 END
```

Listado del programa "Calendario perpetuo". Una aplicación práctica de la introducción de datos de la mano de instrucciones READ/DATA.

minar éstas se ve en la imposibilidad de continuar la lectura.

Como reutilizar los valores de DATA

¿Qué hay que hacer si se quieren utilizar otra vez, dentro del mismo programa, los datos ya leídos por medio de otras instrucciones READ? El BASIC ofrece como solución el comando RESTORE. Su misión es inicializar la lista de constantes almacenadas en sentencias DATA al primer valor de la primera sentencia DATA, con lo que quedan disponibles de nuevo todas las constantes DATA.

El comando RESTORE se puede introducir tanto de forma directa como indirecta. Cada vez que se utiliza un dato almacenado en una sentencia DATA, se incrementa un determinado registro interno de la máquina. Este registro no es más que un puntero o indicador que señala a la primera constante DATA que aún no ha sido utilizada por el programa. Al ejecutar la orden RESTORE, se borra el mencionado registro, con lo que apuntará al primer dato de la primera sentencia DATA.

Algunos dialectos BASIC incluyen una opción muy útil para este comando: permiten especificar un número de línea (o una expresión que calcula un número de líneas existentes en el programa) detrás de la palabra clave RESTORE. Con ello, en lugar de inicializar el puntero a la primera sentencia DATA, lo hará a la línea DATA indicada (o a la primera que se encuentre a partir de la especificada).

Este refinamiento permite empezar a leer de nuevo los datos de cualquier instrucción DATA, incluso de la última del programa, sin más que especificar su número de línea.

El listado adjunto que corresponde a un programa de *calendario perpetuo* constituye un compendio de los conceptos estudiados en el presente capítulo. Al introducir una fecha determinada a través del teclado, el ordenador determinará automáticamente de qué día de la semana se trata.

La línea 60 ilustra la actuación del comando RESTORE al inicializarlo a un determinado número de línea. En este caso el puntero de datos se inicializa a la línea correspondiente al año introducido, mediante el cálculo de la expresión: (200+a−1980). Su resultado será un número comprendido entre 200 y 206, ambos inclusive. Estos son los números de línea de las sentencias DATA que contienen los valores «mágicos», corrrespondientes a cada mes de ese año, que permitirán hallar el nombre del día deseado.

Dicho valor, sumado al día del mes, dará un número comprendido entre 1 y 37. A través de las líneas 110 y 120, el número en cuestión se reducirá a un valor comprendido entre 1 y 7 que, como cabe suponer, corresponde al día de la semana comenzando por el Domingo.

Los nombres de los días de la semana están memorizados en dos instrucciones DATA, localizadas en las líneas 170 y 180; éstas se inicializarán con el comando RESTORE de la línea 130. A partir de ahí, ya no queda más que leer el nombre del día con una sentencia READ y almacenarlo en la variable A$ para su posterior presentación en la pantalla:

```
RUN
FECHA/(EJ.:27,11,1984)
?6,2,1985
EL 6 DEL 2 DE 1985 ES MIERCOLES
█
```

Datos alfanuméricos (I)

Comandos elementales de tratamiento de cadenas


Un ordenador es algo más que una máquina que permite operar con números a gran velocidad. El ordenador también es capaz de procesar otro tipo de información, por ejemplo datos alfanuméricos. Los datos alfanuméricos son secuencias de caracteres (letras, números y símbolos especiales) colocados uno tras otro. En la memoria del ordenador estos caracteres se almacenan mediante sus correspondientes códigos ASCII.

Los caracteres que forman una cadena alfanumérica pueden ser letras mayúsculas o minúsculas, cifras, signos de puntuación, y símbolos algebraicos o de relación. Los datos de tipo alfanumérico pueden manejarse a modo de constantes o por medio de variables. Una constante alfanumérica ha de especificarse encerrando los caracteres que la componen entre comillas. Los siguientes son ejemplos válidos de constantes alfanuméricas:

```
"PEPE"
"U277"
"CARACOL"
"A+2="
```

Una variable alfanumérica tiene el mismo cometido que una variable numérica. En general, una variable es una zona de almacenamiento. En ella se guarda un valor susceptible de ser modificado durante la ejecución de un programa. Al emplear una variable, el ordenador reserva un espacio para su almacenamiento. Tras ello, es posible utilizar dicha zona para almacenar los sucesivos valores asignados a la referida variable.

Para distinguir una variable alfanumérica de otra que no lo es, es necesario terminar su nombre con el símbolo «dólar» ($). A continuación se exponen algunos ejemplos de variables de esta naturaleza:

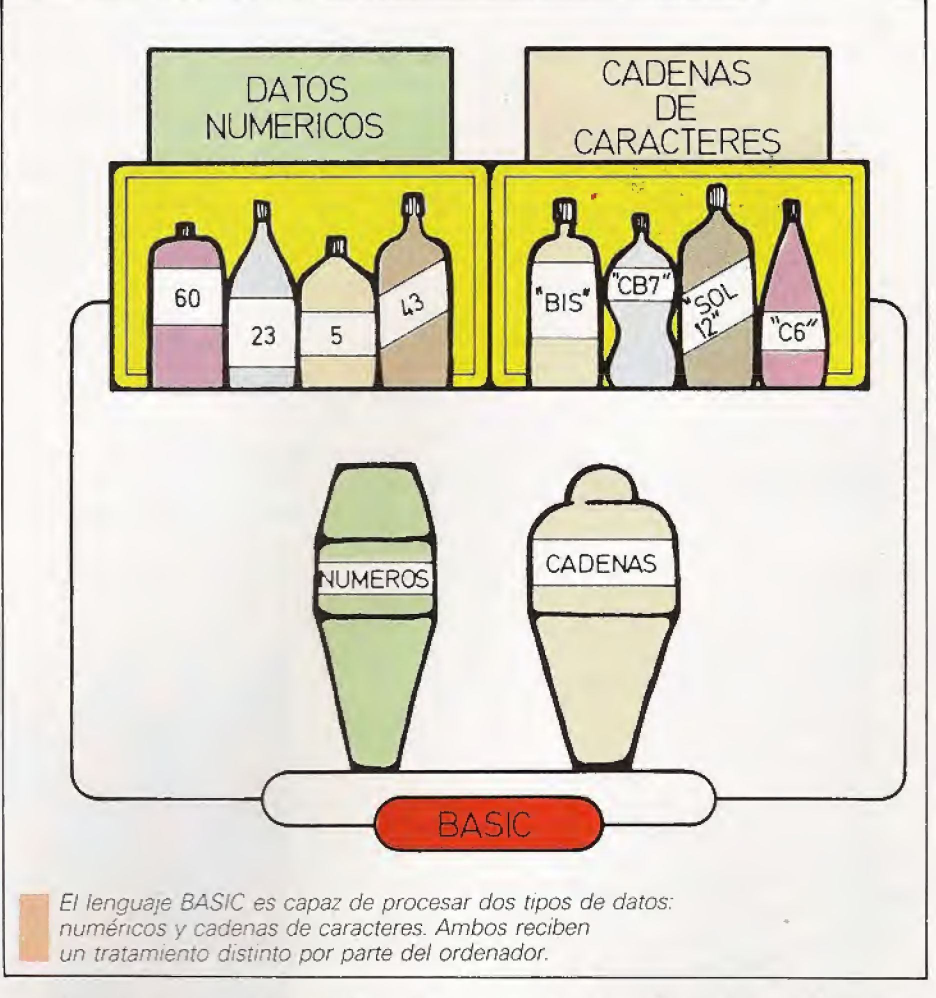
```
A$="ALUMNO"
B$="ES"
C$="JUAN"
D$="UN"
```

Ello nos permitirá realizar algunas combinaciones con la instrucción PRINT. Por ejemplo:

```
PRIN C$,B$,D$,A$
```

Cuyo resultado en la pantalla es el siguiente:

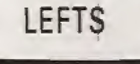
```
JUAN ES UN ALUMNO
```



El lenguaje BASIC es capaz de procesar dos tipos de datos: numéricos y cadenas de caracteres. Ambos reciben un tratamiento distinto por parte del ordenador.

Es preciso diferenciar los datos alfanuméricos de los numéricos, puesto que son de distinta naturaleza y son tratados por la máquina de distinta forma. Así, por ejemplo, una asignación como la siguiente:

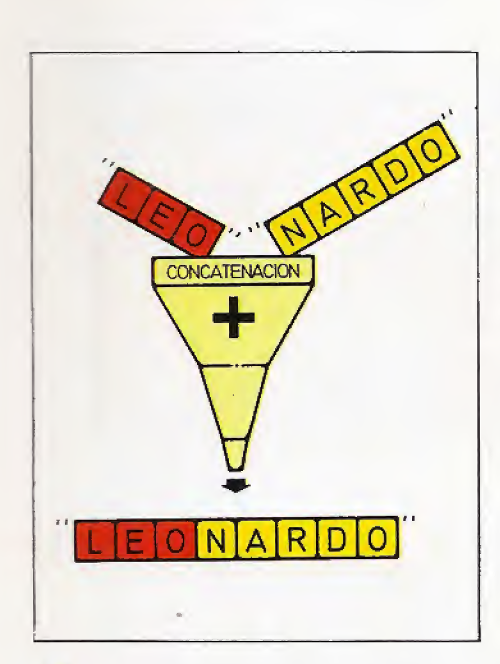
```
B$=234
```



Extrae los N caracteres situados más a la izquierda de la cadena sobre la que actúa. Se trata de una función, de ahí que deba actuar en combinación con un comando.

Formato: LEFT$(<argumento>, N)
<argumento>: cadena de caracteres afectada.
N: número natural.

Ejemplos: PRINT LEFT$ ("CASA", 2)
LET B$=LEFT$ (A$, 5)



La concatención de cadenas de caracteres es la operación básica realizable sobre datos de tipo alfanumérico. Consiste, sencillamente, en la unión de las cadenas componentes.

sería rechazada con la presencia de un mensaje de error del tipo «TYPE MISMATCH» o similar. Con ello, el ordenador indica que la asignación es incorrecta, ya que se intenta mezclar dos tipos distintos de datos. Tal incorrección puede arreglarse poniendo comillas en el lugar adecuado:

```
B$="234"
```

La nueva asignación será aceptada por el ordenador sin ningún problema. No obstante, hay que tener en cuenta que la variable B$ no tiene asignado el número 234, sino la secuencia de caracteres 2, 3, 4: lo que significa que no se pueden realizar operaciones algebraicas con dicha variable.

Fraccionamiento de cadenas

La posibilidad de unir dos o más cadenas para formar una nueva (concatenación de cadenas), permite construir distintas combinaciones partiendo de varias cadenas componentes. El siguiente caso, opuesto al anterior, radica en la posibilidad de extraer fragmentos de una cadena; esto es, separar una parte o subconjunto de la misma. Para ello el BASIC dispone de tres funciones: LEFT$, RIGHT$ y MID$.

La primera de ellas resulta adecuada para extraer los elementos situados a la izquierda de la cadena de caracteres (no hay que olvidar que la palabra LEFT significa izquierda en inglés).

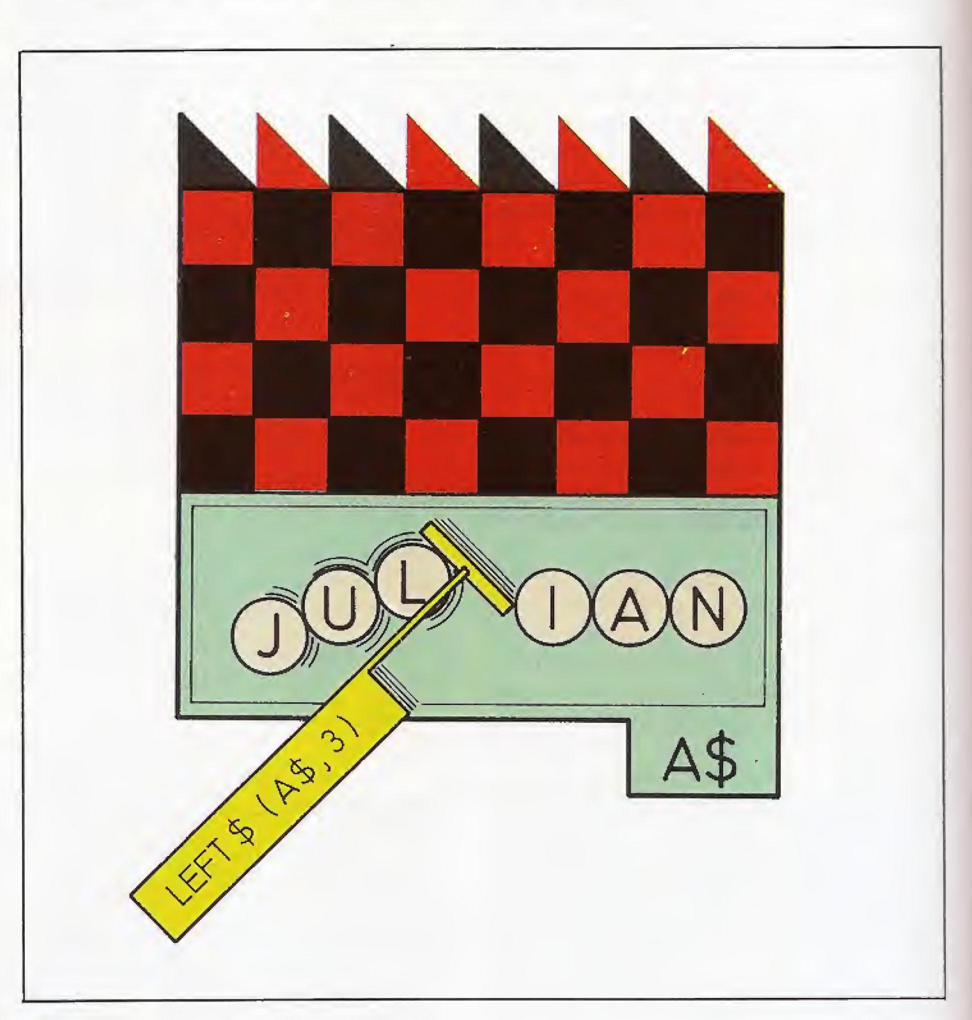
El formato de esta función es el siguiente:

LEFT$(<argumento>, N)

La zona <argumento> contiene la cadena sobre la que se realiza la operación. Esta puede expresarse como dato constante, o bien, puede incluir a una variable representativa de la cadena de caracteres afectada. A su vez, N es un número natural que especifica el número de caracteres a extraer de la cadena, por supuesto, a partir de su extremo izquierdo. Las siguientes instrucciones incluyen la función definida:

```
PRINT LEFT$("SOFTWARE",5)
A$=LEFT$("CASA",2)
C$=LEFT$(S$,3)
```

Hay que hacer notar que LEFT$ es una función y, por lo tanto, ha de utilizarse asociada a un comando. De lo contrario, el ordenador no sabría qué hacer con el resultado obtenido. Por ejemplo, la función LEFT$ puede utilizarse dentro de una instrucción PRINT:



La función LEFT$ resulta adecuada para extraer un determinado número de caracteres localizados en la zona izquierda de una cadena.

TABLA DE CONVERSION				
Ordenador	Concatenación	LEFT$	RIGHT$	MID$
	<c1>+<c2>	LEFT$ (<c>, <n>)	RIGHT$ (<c>, <n>)	MID$ (<c>, <p>, <n>)
AMSTRAD	<c1>+<c2>	LEFT$ (<c>, <n>)	RIGHT$ (<c>, <n>)	MID$ (<c>, <p>, <n>)
APPLE II (APPLESOFT)	<c1>+<c2>	LEFT$ (<c>, <n>)	RIGHT$ (<c>, <n>)	MID$ (<c>, <p>, <n>)
APRICOT (M-BASIC)	<c1>+<c2>	LEFT$ (<c>, <n>)	RIGHT$ (<c>, <n>)	MID$ (<c>, <p>, <n>)
ATARI	—	—	—	<c> (<p>, <q>)
CBM 64	<c1>+<c2>	LEFT$ (<c>, <n>)	RIGHT$ (<c>, <n>)	MID$ (<c>, <p>, <n>)
DRAGON	<c1>+<c2>	LEFT$ (<c>, <n>)	RIGHT$ (<c>, <n>)	MID$ (<c>, <p>, <n>)
EQUIPOS MSX	<c1>+<c2>	LEFT$ (<c>, <n>)	RIGHT$ (<c>, <n>)	MID$ (<c>, <p>, <n>)
HP-150	<c1>+<c2>	LEFT$ (<c>, <n>)	RIGHT$ (<c>, <n>)	MID$ (<c>, <p>, <n>)
IBM PC	<c1>+<c2>	LEFT$ (<c>, <n>)	RIGHT$ (<c>, <n>)	MID$ (<c>, <p>, <n>)
MPF	<c1>+<c2>	LEFT$ (<c>, <n>)	RIGHT$ (<c>, <n>)	MID$ (<c>, <p>, <n>)
NCR DM-V (MS-BASIC)	<c1>+<c2>	LEFT$ (<c>, <n>)	RIGHT$ (<c>, <n>)	MID$ (<c>, <p>, <n>)
NEW BRAIN	<c1>+<c2>	LEFT$ (<c>, <n>)	RIGHT$ (<c>, <n>)	MID$ (<c>, <p>, <n>)
ORIC	<c1>+<c2>	LEFT$ (<c>, <n>)	RIGHT$ (<c>, <n>)	MID$ (<c>, <p>, <n>)
SHARP MZ-700 (MZ-BASIC)	<c1>+<c2>	LEFT$ (<c>, <n>)	RIGHT$ (<c>, <n>)	MID$ (<c>, <p>, <n>)
SINCLAIR QL	&	—	—	<c> (<p> TO <q>)
SPECTRAVIDEO	<c1>+<c2>	LEFT$ (<c>, <n>)	RIGHT$ (<c>, <n>)	MID$ (<c>, <p>, <n>)
ZX-SPECTRUM	<c1>+<c2>	—	—	<c> (<p> TO <q>)

<c1>, <c2>, <c>: Cada alfanumérica con la que se ha de realizar la operación. <n>: Número de caracteres a extraer. <p>: Número de orden del primer carácter a extraer. <q>: Número de orden del último carácter a extraer.

```
PRINT LEFT$("JUANJOSE",4) <CR>
JUAN
█
```

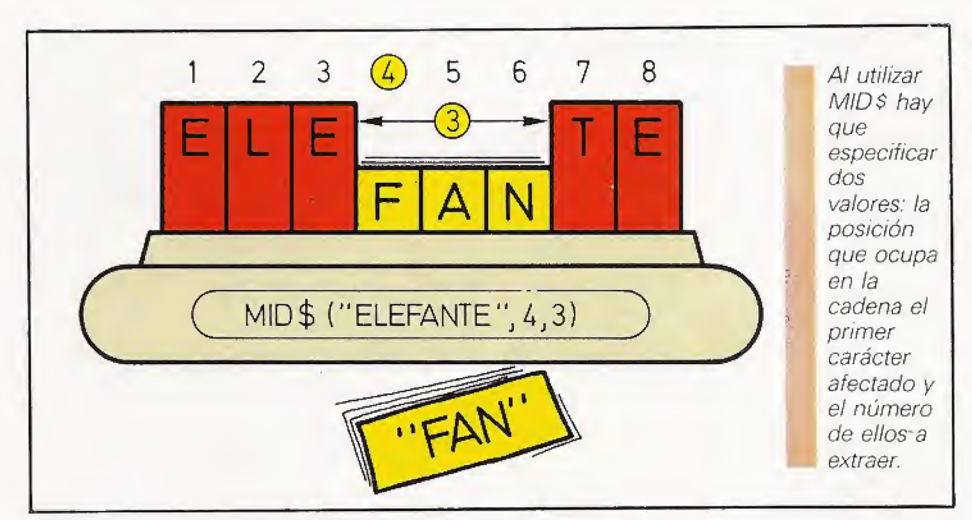
Como se ve, la función LEFT$ se emplea para extraer los cuatro caracteres situados más a la izquierda de la cadena indicada. El número N con el que se define el número de caracteres a extraer de la cadena, puede ser sustituido también por una variable; en tal caso, se extraerá el número de caracteres que se-

RIGHT$

Extrae los N caracteres situados más a la derecha de la cadena.

Formato: RIGHT$(<argumento>, N)
<argumento>: cadena de caracteres.
N: número natural.

Ejemplos: PRINT RIGHT$(B$, 4)
B$=RIGHT$("CASUCHA", 3)



Al utilizar MID$ hay que especificar dos valores: la posición que ocupa en la cadena el primer carácter afectado y el número de ellos a extraer.

ñale el contenido de la variable en cuestión.

Desde luego, también es posible extraer un conjunto de caracteres situados a la derecha de una determinada cadena. Para realizar esta función, hay que recurrir a la función RIGHT$. Su formulación es la siguiente:

RIGHT$(<argumento>,N)

De nuevo, en el campo denominado argumento se especifica la cadena, en forma de constante o variable, mientras que N es el número de caracteres que se desea extraer. Por ejemplo:

```
PRINT RIGHT$("TITO",2)
B$=RIGHT(B$,5)
```

Ambas son instrucciones que hacen un uso correcto de la función RIGHT$. Al igual que LEFT$, RIGHT$ es también una función, por lo que las consideraciones que se hicieron anteriormente son extensivas a este caso. Por ejemplo, una instrucción como la que sigue:

```
RIGHT$("TIO",2)
```

obtendría como respuesta un mensaje de error: no hay que perder de vista que una función debe operar siempre asociada a un comando.



También es posible representar por medio de una variable el número de caracteres a recoger de la cadena.

Desde luego, ambas funciones permiten un amplio abanico de posibilidades, pero dejan abierta una cuestión: ¿es posible extraer un conjunto de caracteres que no estén situados a los extremos de la cadena? La respuesta es sí; para ello hay que emplear una nueva función denominada MID$. Su formato es el siguiente:

MID$(<argumento>,N,M)

La cadena sobre la que ha de actuar se especifica en la zona <argumento>. N indica el carácter a partir del cual se desea efectuar la extracción, y M señala el número de caracteres a extraer.

La función opera colocando el punto de «corte» de la cadena en el carácter situado en la posición N, contada a partir de la izquierda, y extrae los M caracteres siguientes. Veamos un ejemplo práctico.

```
PRINT MID$("CADENA",3,2) <CR>
DE
█
```

El punto de corte se sitúa a partir del tercer carácter, y se extraen los dos caracteres que siguen.

Esta función, empleada adecuadamente, permite extraer por separado todos y cada uno de los caracteres de una cadena alfanumérica.

Una de las aplicaciones del tratamiento de cadenas es la comprobación de mensajes destinados a mantener un diálogo hombre-máquina. Quizás, el caso más sencillo sea la simple comprobación de si la respuesta dada a una pregunta es SI o NO. Veamos esta posibilidad:

```
10 INPUT"DESEA VER EL RESULTADO";B$
20 IF B$="SI" THEN GOTO 100
30 IF B$="NO" THEN GOTO 200
40 PRINT "ERROR"
50 GOTO 10
100 PRINT "RESULTADO";2+2
200 END
```

Esta es una zona de decisión muy frecuente en los programas BASIC. A la pregunta señalada en la línea 10, hay que responder afirmativa o negativamente. Las instrucciones 20 y 30 comprueban si la respuesta es SI o NO y bifurcan la ejecución según corresponda.

MID$

Extrae un grupo de M caracteres de la cadena indicada, tomados a partir del carácter que ocupa la posición N (inclusive).

Formato: MID$(<argumento>, N, M)
<argumento>: cadena de caracteres sobre la que actúa.
N y M: números naturales.

Ejemplos: PRINT MID$("CADENA", 3, 2)
B$=MID$(C$, 5, 5)

Datos alfanuméricos (II)

Números, caracteres y sus relaciones



El tratamiento de cadenas alfanuméricas confiere una particular potencia a los ordenadores al capacitarlos para manipular información no numérica. Este hecho establece una de las diferencias fundamentales entre un ordenador y una máquina de calcular programable. La actividad de las calculadoras se reduce al tratamiento de información numérica y sólo en algunos casos permiten la generación de determinados mensajes, aunque no su tratamiento.

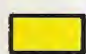
En el capítulo anterior se han estudiado ya algunas funciones BASIC para el tratamiento de cadenas de caracteres; sin embargo, quedan aún muchas funciones en el tintero. Para proseguir con su estudio, partiremos en esta ocasión de un ejemplo «gramatical».

Suponga que a partir de un verbo, dado en forma de infinitivo, se pretende separar por un lado la terminación verbal y por otro su raíz. Esta operación no resultará difícil si se trata de un verbo cuyo número de caracteres es conocido. Por ejemplo, imagine el verbo «sacar»:

```
10 PRINT RIGHT$("SACAR",2)
20 PRINT LEFT$("SACAR",3)
30 END
■
```

RIGHT$ (A$,2) 



 LEFT$ (A$,3)

RIGHT$(A$,2) 



 LEFT$ (A$,3)

Actuación de las funciones LEFT$ y RIFHT$ sobre una cadena de caracteres. En el primer caso se extrae correctamente la raíz y la desinencia de la palabra operada, mientras que en el segundo la raíz aparece truncada.

El resultado de este simple programa es el que aparece en la pantalla:

```
RUN
AR
SAC
■
```

No se ha presentado problema alguno ya que, en este caso, el verbo a tratar era conocido de antemano. No obstante, el programa puede complicarse algo más, en orden a permitir la introducción de cualquier verbo:

```
10 INPUT A$
20 PRINT RIGHT$(A$,2)
30 PRINT LEFT$(A$,3)
40 STOP
■
```

Ahora, el dato (verbo) no es fijo, sino que será leído por efecto de la instrucción INPUT que aparece en la línea 10.

¿Qué ocurrirá si el dato introducido es un verbo cuyo infinitivo tiene más o menos de cinco caracteres? Por ejemplo, «correr»:

```
RUN
?CORRER <CR>
ER
COR
■
```

El resultado no es correcto, falta una letra en la zona de «raíz». Es evidente que para que el programa resulte eficaz éste debe ser capaz de medir la longitud del dato de entrada (el verbo) y fraccionarlo en consecuencia.

El lenguaje BASIC dispone de una función que permite evaluar la longitud de una cadena de caracteres. Esta es LEN, cuyo formato general coincide con:

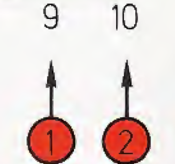
LEN (d<cadena>)

Esta función da como resultado un número natural, que coincide con el número de caracteres que constituyen la cadena especificada en el argumento (incluyendo los espacios en blanco).

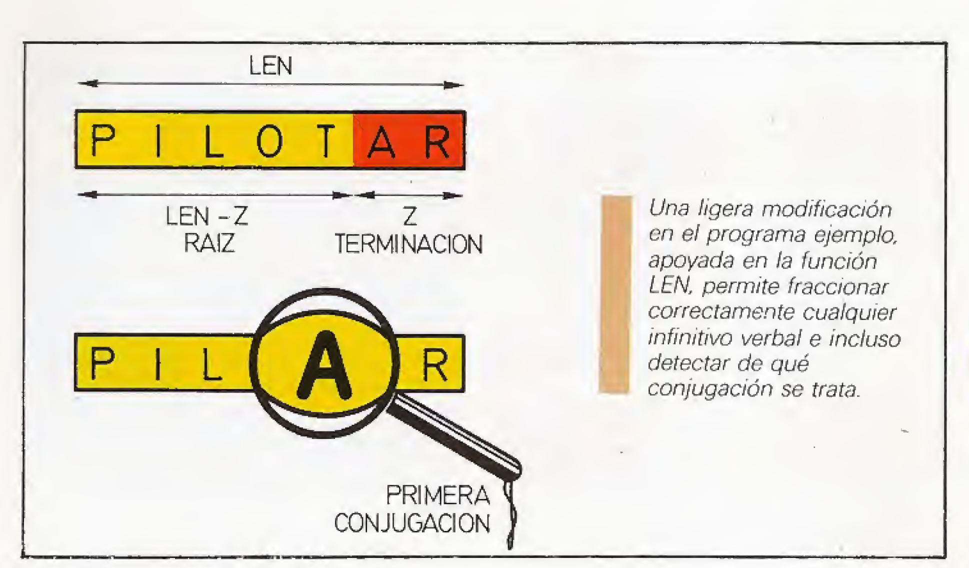
Hay que recordar que LEN es una fun-

A S O M B R O S O

1 2 3 4 5 6 7 8 9 10



La función LEN permite medir la longitud de una cadena alfanumérica, incluyendo los espacios en blanco. En el caso de la figura, si la cadena coincidiera con la palabra «asombroso» su longitud sería 9, mientras que si incluye un espacio en blanco al final de la misma esta longitud será de 10 caracteres.



Una ligera modificación en el programa ejemplo, apoyada en la función LEN, permite fraccionar correctamente cualquier infinitivo verbal e incluso detectar de qué conjugación se trata.

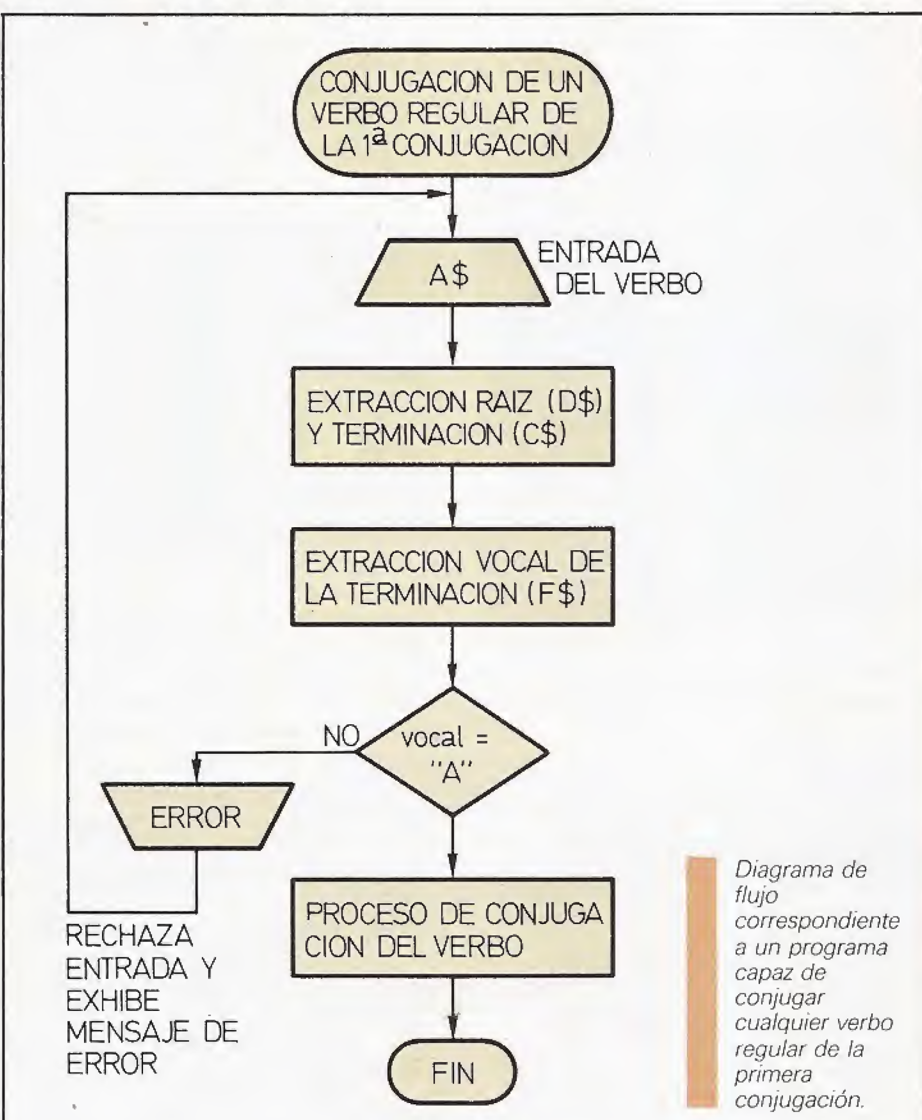
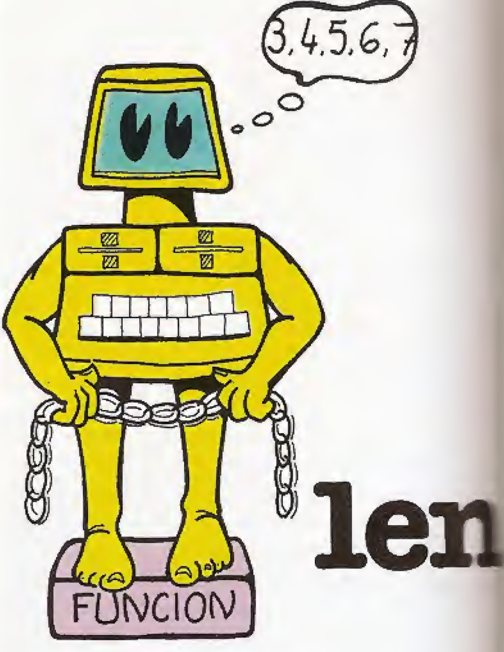


Diagrama de flujo correspondiente a un programa capaz de conjugar cualquier verbo regular de la primera conjugación.

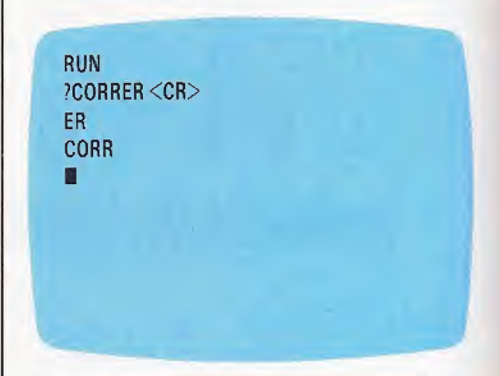


ción y, por lo tanto, será preciso utilizarla asociada a un comando.

Veamos cuál será el nuevo aspecto del programa ejemplo, una vez introducida la función LEN:

```
10 INPUT A$
20 LET B=LEN(A$)
30 LET C$=RIGHT$(A$,2)
40 LET D$=LEFT$(A$,B—2)
50 PRINT C$
60 PRINT D$
70 END
```

La ejecución del programa conduce al siguiente resultado:



En efecto, el problema ha dejado de existir. El programa admite ahora cualquier verbo, sea cual fuere su longitud o número de caracteres. La razón de ello estriba en que la terminación verbal

siempre coincide con las dos últimas letras (RIGHT$(A$, 2). Por lo tanto, la raíz ha de estar compuesta por las restantes letras, cuyo número será igual a B—2: la longitud total calculada por medio de la función LEN, menos las dos letras de la terminación verbal.

El código ASCII

Las cadenas de caracteres se almacenan en el ordenador en código ASCII. Como ya es sabido, el código ASCII es un método de representación binario adecuado para codificar a los diversos caracteres alfanuméricos y símbolos especiales. El código de cada carácter ha de almacenarse en ocho bits, de ahí que sólo pueda variar entre cero y 255 en su equivalencia decimal (2^8=256 posibles representaciones). Sin embargo, no suelen utilizarse todos los posibles códigos.

Los restantes códigos se reservan para otros cometidos. Por ejemplo, para la representación de caracteres gráficos especiales y caracteres de control. Mediante éstos se canalizan órdenes destinadas al propio ordenador y a los dispositivos periféricos asociados al mismo.

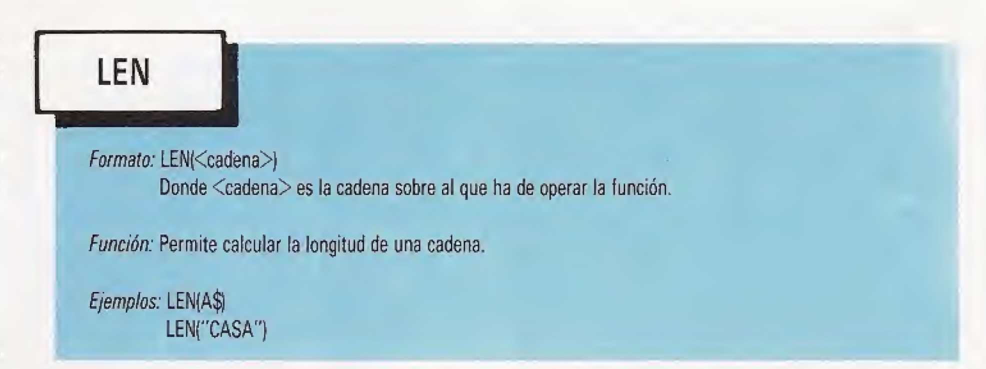
El código ASCII es compartido por distintos ordenadores por lo que respecta a la representación de los caracteres alfabéticos, numéricos y algunos signos de puntuación; no obstante, suele diferir en lo relativo a los caracteres de control. Aunque bien es cierto que algunos códigos pueden coincidir; tal es el caso del retroceso de carro (13), avance de línea (10) y ESCAPE (27). De todas formas, lo mejor es consultar siempre la tabla de códigos de cada equipo.

El lenguaje BASIC permite realizar ciertas manipulaciones con los códigos ASCII. Para empezar, es posible definir una cadena constituida por un carácter expresado por medio de su código ASCII. La función adecuada es CHR$( ), cuyo formato genérico adopta el siguiente aspecto:

CHR$(<número>)

coincidiendo el argumento <número> con el código ASCII del carácter deseado. Por ejemplo:

```
PRINT CHR$(48)
```



LEN

Formato: LEN(<cadena>)
Donde <cadena> es la cadena sobre al que ha de operar la función.

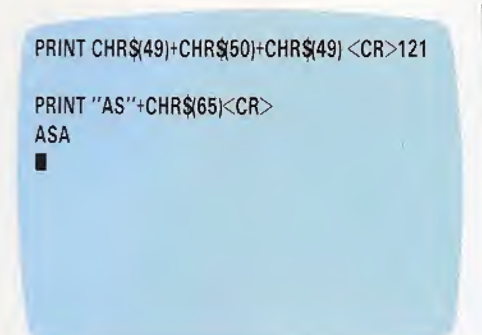
Función: Permite calcular la longitud de una cadena.

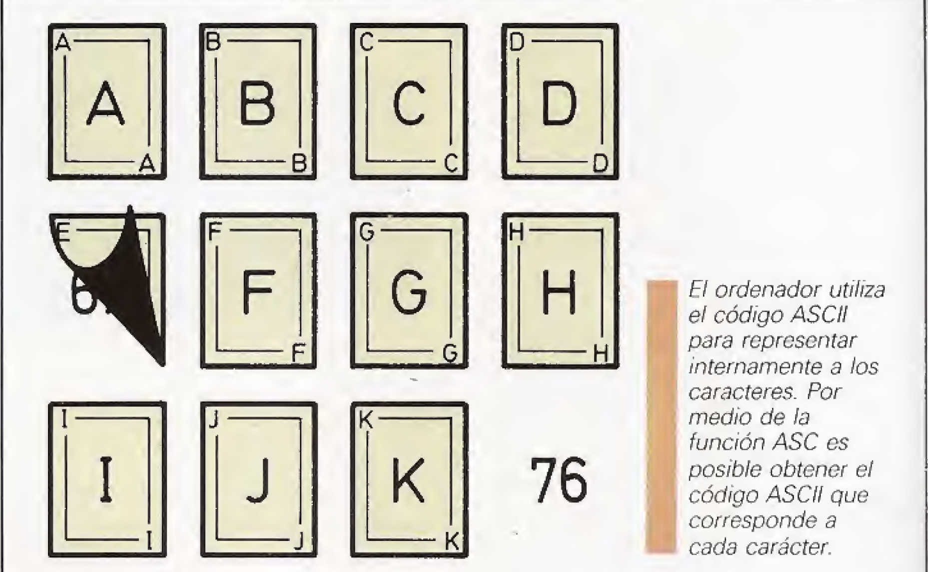
Ejemplos: LEN(A$)
LEN("CASA")

```
B$=CHR$(68)
PRINT CHR$(A)
LET B$=CHR$(A+2)
```

También es posible encadenar varias de esta funciones para construir una cadena más larga; por ejemplo:

Modificando el organigrama de la figura anterior es posible realizar un programa capaz de conjugar cualquier verbo regular.





El ordenador utiliza el código ASCII para representar internamente a los caracteres. Por medio de la función ASC es posible obtener el código ASCII que corresponde a cada carácter.

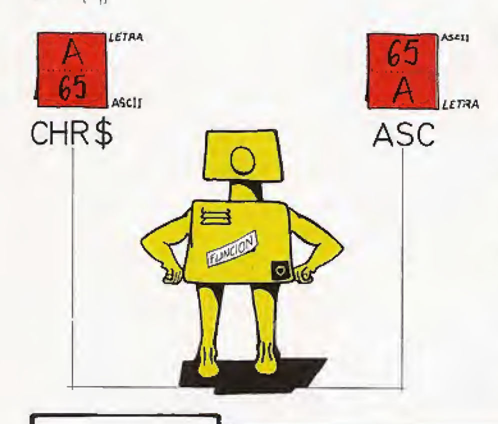
Otra función BASIC de esta categoría es ASC, adecuada para obtener el código ASCII del primer carácter de una cadena dada. Su formulación es la siguiente:

ASC (<cadena>)

El argumento <cadena> debe contener a la cadena alfanumérica afectada. La respuesta a la función ASC coincidirá con el código ASCII correspondiente a su primer carácter, empezando por la izquierda.

Por ejemplo:

```
PRINT ASC("A")
PRINT ASC("ALTA")
PRINT ASC(B$)
A=ASC(L$)
```



A partir de las funciones definidas, puede confeccionarse un programa capaz de operar la conversión de un número binario a decimal.

En primer lugar es oportuno conocer la «receta» para efectuar la conversión de binario a decimal. Para empezar hay que tomar cada uno de los dígitos del número binario, a partir del situado más a la derecha, y multiplicarlo por la potencia de dos que corresponda al orden que ocupa, empezando por la potencia cero. El valor de cada una de las respectivas potencias de 2 es lo que se llama *peso* del dígito en cuestión.

Así, por ejemplo, el número binario 10110101 presenta la siguiente correspondencia entre bits y pesos:

1	0	1	1	0	1	0	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

Su homólogo en el sistema de numeración decimal se obtiene, sencillamente, sumando los productos de cada dígito por su peso respectivo. Esto es:

$1\times2^0+0\times2^1+1\times2^2+0\times2^3+1\times2^4+1\times2^5+0\times2^6+1\times2^7=1\times1+0+1\times4+0+1\times16+1\times32+0+1\times128=1+4+16+32+128=181$

En definitiva, el número binario 10110101 equivale al 181 en el sistema de numeración decimal.

Y pasemos ahora al programa adecuado para operar esta conversión de forma automática.

En primer lugar, será preciso introducir la cadena de ceros y unos y evaluar su longitud. De ello se ocuparán las siguientes instrucciones:

```
10 INPUT "EL NUMERO BINARIO A CONVERTIR ES:";A$
20 B=LEN(A$):REM NUMERO DE DIGITOS DE LA CADENA
```

Tras ello es necesario extraer cada uno de los dígitos o bits de la cadena y comprobar si es cero o uno. Por supuesto, si el dígito no coincide con 0 ó 1, habrá que mostrar un mensaje de error. A continuación, hay que multiplicar cada dígito por su peso correspondiente.

Todo ello correrá a cargo de un bucle. Este extraerá los caracteres, empezando por el situado más a la izquierda; en consecuencia, el peso del primer carácter será dos elevado a la longitud de la cadena menos uno. El siguiente será dos elevado a la longitud de la cadena menos dos y así sucesivamente.

CHR$

Formato: CHR$(<número>)
<número>. Es un número natural que no ha de ser mayor de 255.

Función: Produce una cadena de un único carácter cuyo código ASCII corresponde al <número> especificado.

Ejemplos: CHR$(32)
CHR$(7)

```
 30 FOR I=1 TO B
 40 B$=MID$(A$,I,1)
 50 C=ASC(B$)
 60 IF C<48 OR C>49 THEN PRINT "ERROR":GOTO 10
 70 LET C=C—48
 80 LET N=C*2 ↑ (B—I)+N
 90 NEXT I
100 PRINT "LA SOLUCION ES:";N
110 END
```

La instrucción de la línea 70 convierte la cadena «0» o la cadena «1» en el número 0 ó 1 según el caso. Para entender su actuación hay que recordar que el código ASCII del «0» es 48 y el del «1» es 49. En consecuencia, al restar el valor 48 del código en cuestión, se obtendrá el número correspondiente.

Una vez agrupadas las distintas zonas, el programa conjunto ofrecerá el siguiente aspecto:

```
  5 LET N=0
 10 INPUT "EL NUMERO BINARIO A CONVERTIR ES:";A$
 20 B=LEN(A$): REM NUMERO DE DIGITOS DE LA CADE
    NA
 30 FOR I=1 TO B
 40 B$=MID$(A$,I,1)
 50 C=ASC(B$)
 60 IF C<48 OR C>49 THEN PRINT "ERROR":GOTO 5
 70 LET C=C—48
 80 LET N=C*2 ↑ (B—I)+N
 90 NEXT I
100 PRINT "LA SOLUCION ES:";N
110 END
```

Desde luego, el programa es susceptible de incorporar modificaciones a voluntad del usuario. Incluso el método adoptado podría haber sido otro. El hecho de optar por este procedimiento de conversión se debe a que es posible operar conversiones de números expresados en una base distinta de la binaria; por ejemplo, hexadecimal u octal. De ahí que la adaptación del programa para que sea capaz de realizar otro tipo de conversión resulte de lo más inmediato.

Tratamiento de cadenas en el BASIC Sinclair

Quizás los únicos ordenadores que presentan diferencias sustanciales en la forma de tratar a las cadenas alfanuméricas son los de la firma Sinclair (el popular ZX-SPECTRUM, por ejemplo). Estos equipos no hacen uso de ninguna de las funciones habituales (LEFT$, RIGHT$ y MID$), sino que recurren al empleo de una función propia. Esta es TO, cuyo formato es el siguiente:

<argumento>(N TO M)

Al igual que en el caso genérico, la

TABLA DE CONVERSION

Ordenador	LEN	CHR$	ASC
	LEN (<c>)	CHR$ (<n>)	ASC (<c>)
AMSTRAD	LEN (<c>)	CHR$ (<n>)	ASC (<c>)
APPLE II (APPLESOFT)	LEN (<c>)	CHR$ (<n>)	ASC (<c>)
APRICOT (M-BASIC)	LEN (<c>)	CHR$ (<n>)	ASC (<c>)
ATARI	LEN (<c>)	CHR$ (<n>)	ASC (<c>)
CBM 64	LEN (<c>)	CHR$ (<n>)	ASC (<c>)
DRAGON	LEN (<c>)	CHR$ (<n>)	ASC (<c>)
EQUIPOS MSX	LEN (<c>)	CHR$ (<n>)	ASC (<c>)
HP-150	LEN (<c>)	CHR$ (<n>)	ASC (<c>)
IBM PC	LEN (<c>)	CHR$ (<n>)	ASC (<c>)
MPF	LEN (<c>)	CHR$ (<n>)	ASC (<c>)
NCR DM-V (MS-BASIC)	LEN (<c>)	CHR$ (<n>)	ASC (<c>)
NEW BRAIN	LEN (<c>)	CHR$ (<n>)	ASC (<c>)
ORIC	LEN (<c>)	CHR$ (<n>)	ASC (<c>)
SHARP MZ-700 (MZ-BASIC)	LEN (<c>)	CHR$ (<n>)	ASC (<c>)
SINCLAIR QL	LEN (<c>)	CHR$ (<n>)	CODE (<c>)
SPECTRAVIDEO	LEN (<c>)	CHR$ (<n>)	—
ZX-SPECTRUM	LEN (<c>)	CHR$ (<n>)	CODE (<c>)

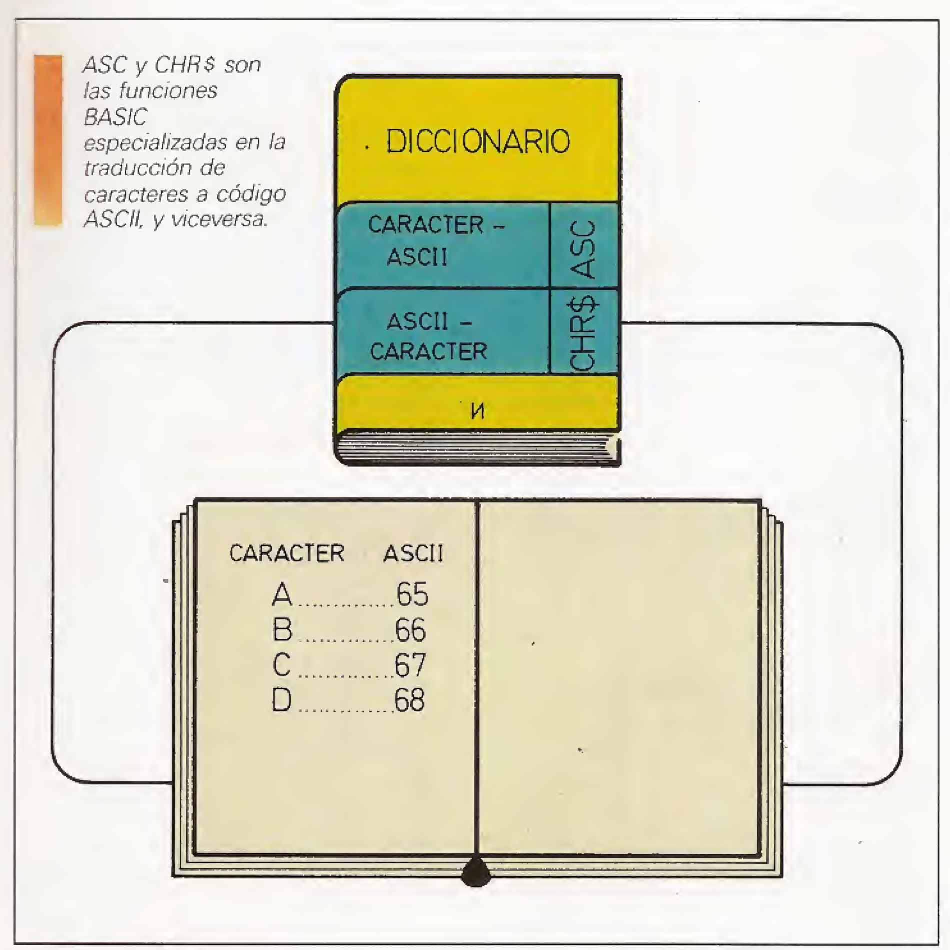
<c>: Cadena de caracteres o variable alfanumérica. <n>: Número coincidente con el código ASCII de un carácter.

ASC

Formato: ASC(<cadena>)
<cadena> es la cadena de caracteres sobre la que opera la función.

Función: Esta función da como resultado un número que corresponde al código ASCII del primer carácter de la cadena sobre la que opera.

Ejemplos: ASC(A$)
ASC("S")



ASC y CHR$ son las funciones BASIC especializadas en la traducción de caracteres a código ASCII, y viceversa.

zona de argumento incluye a la cadena sobre la que debe actuar la función. N señala el punto inicial o de «corte», a partir del cual se extraerán los caracteres. Por último, M indica el último carácter que forma parte de la subcadena a extraer.

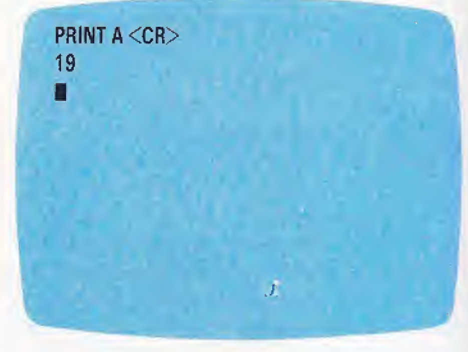
Las funciones de extracción de caracteres por la izquierda y por la derecha de la cadena (LEFT$ y RIGHT$) son perfectamente operables con esta nueva sintaxis. Al omitir el valor N en la expresión general, el equipo extraerá los caracteres situados a la izquierda de la cadena, hasta llegar al que ocupa la orden M. En efecto, esta formulación equivale a la propia de la función LEFT$. Así, por ejemplo, la función LEFT$(A$, X) se representará en el BASIC de Sinclair de la siguiente forma: A$(TO X).

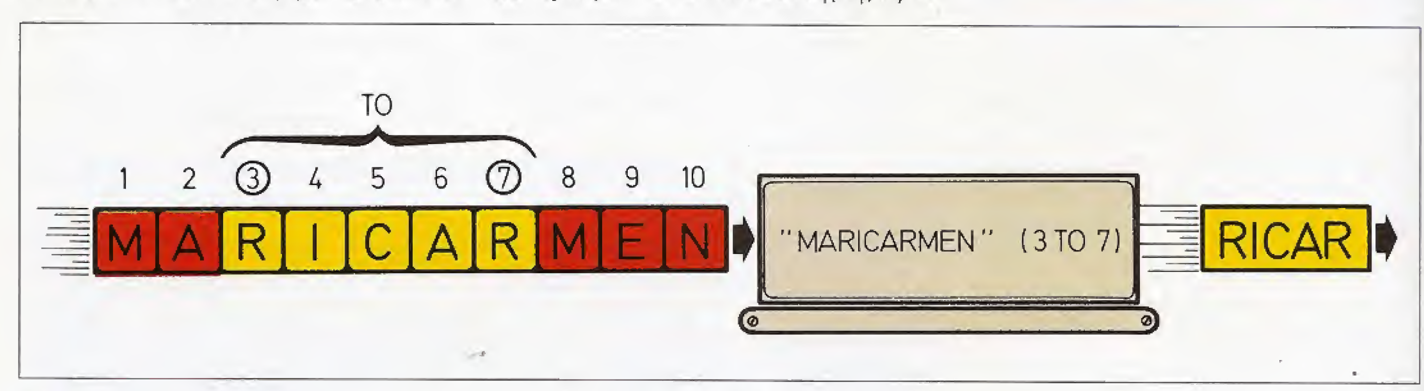
Algo semejante ocurre al omitir el valor final (M). Por ejemplo, la función A$(N TO), construirá una cadena constituida por los caracteres A$ situados a partir de la posición N hasta el final.

Existe aún otra salvedad, esta vez propia del BASIC Sinclair que incorpora el Sinclair QL. En el texto se ha mostrado la imposibilidad de operar con datos de distinto tipo (numéricos y alfanuméricos) mezclados. El QL de Sinclair evita este problema, adecuando automáticamente el tipo del dato al de la variable que recibe la asignación. En el siguiente ejemplo:

```
A=17+"2"
```

La variable numérica A guardará la suma del dato numérico 17 y el dato numérico 2. Lo que hace es convertir previamente el carácter alfanumérico "2" a su correspondiente valor numérico (2). Así pues, además de no producir un mensaje de error, su contenido será el siguiente:





En los ordenadores de la firma Sinclair —como el popular SPECTRUM—, las funciones habituales para la extracción de caracteres son reemplazadas por la función genérica TO. Los dos valores que la acompañan identifican al primer y último caracteres que hay que extraer.

Trabajando con cadenas

Funciones evolucionadas para manipular datos alfanuméricos



El BASIC es un lenguaje que ofrece grandes posibilidades para el tratamiento de la información alfanumérica. Frente a otros lenguajes de propósito más específico en uno u otro ámbito de la actividad informática, el BASIC ha rechazado la especialización y se emplea en un amplio abanico de aplicaciones. Desde las más triviales, como por ejemplo los juegos, hasta las más serias, como puedan ser las aplicaciones de gestión y enseñanza. En muchas de estas aplicaciones, la facilidad para el manejo de cadenas de caracteres es una de las facetas que hacen del BASIC un lenguaje insustituible y de amplia aplicación.

En este capítulo se estudiarán algunas de las herramientas que ofrece el lenguaje BASIC para la manipulación de cadenas de caracteres. En el apartado de funciones, hay que hablar de las encargadas de la conversión de cadenas a números y de números a cadenas; ampliamente utilizadas y que se encuentran en casi todos los dialectos BASIC. Las restantes funciones a estudiar son específicas de ciertos dialectos más potentes. Estas últimas resultan de gran utilidad al facilitar el tratamiento de cadenas, aunque no son realmente imprescindibles.

De números a cadenas

En ocasiones es preciso transformar cadenas de caracteres en números y viceversa. Esta transformación se hace necesaria para permitir el tratamiento de determinados datos numéricos. Con ello se consigue, por ejemplo, la extracción de determinadas cifras aisladas de una forma cómoda y eficaz. La representación de números como cadenas de caracteres ayuda a almacenarlos en archivos y contribuye a economizar memoria en el ordenador.

Consideremos en primer lugar la operación de convertir un dato numérico en una cadena de caracteres. Para ello es necesario emplear la función STR$. Esta se encargará de realizar la transformación adecuada para conseguir dicho resultado. Su aspecto general es el siguiente:

STR$ (<numero>)

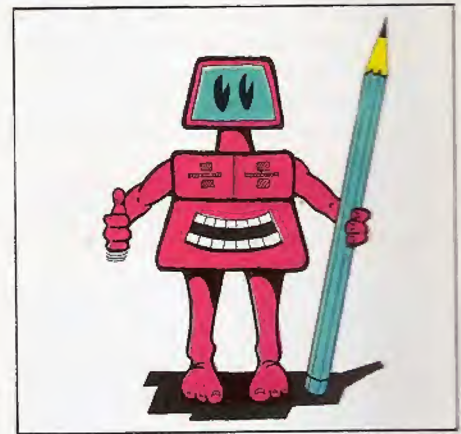
En donde <número> representa a la constante o variable numérica que se desea convertir en cadena de caracteres. En consecuencia, las siguientes son todas ellas formulaciones válidas de la función STR$

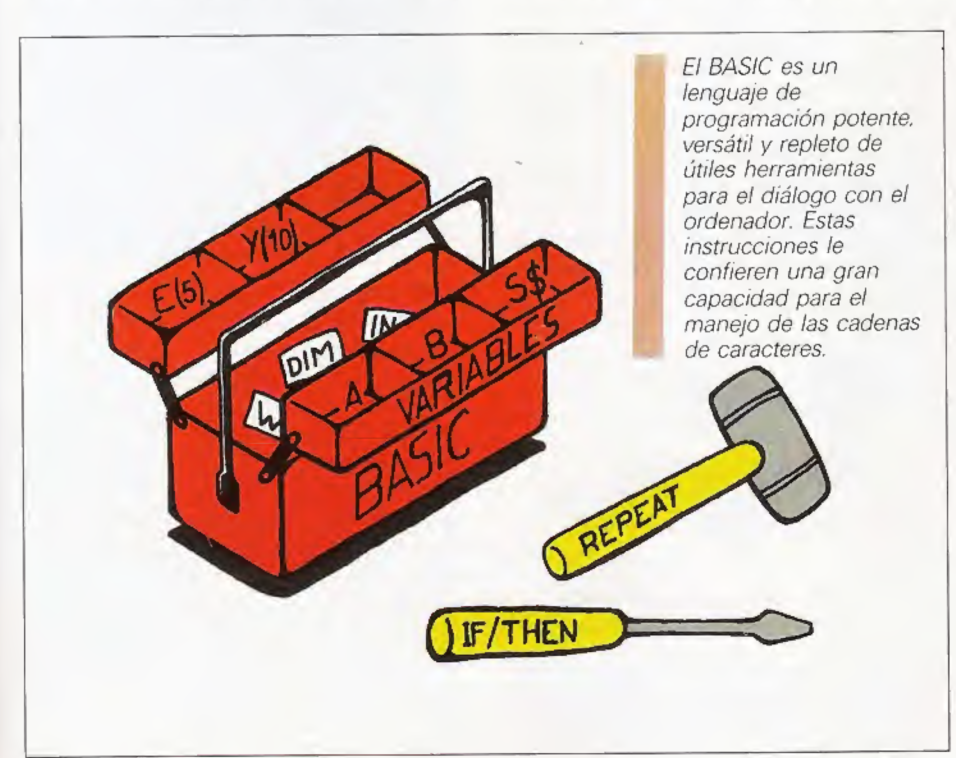
```
STR$ (128)
STR$ (274)
STR$ (A)
STR$ (N)
```

La función que nos ocupa actúa sobre un dato numérico, convirtiéndolo en una cadena cuyos caracteres coinciden con las cifras que componen el dato de partida. La diferencia entre uno y otro formato no se manifiesta en sus correspondientes representaciones en pantalla. No obstante, sí cabe señalar que los nú-



Frente a otros lenguajes de propósito más específico, el BASIC presenta una gran aptitud para un amplio abanico de aplicaciones. Una de las bazas que apoyan tal versatilidad reside en su capacidad para el tratamiento de cadenas de caracteres.





El BASIC es un lenguaje de programación potente, versátil y repleto de útiles herramientas para el diálogo con el ordenador. Estas instrucciones le confieren una gran capacidad para el manejo de las cadenas de caracteres.

STR$

Función: Convierte un número en una cadena de caracteres.

Formato: STR$(<número>)
<número>: número que se desea convertir en cadena.

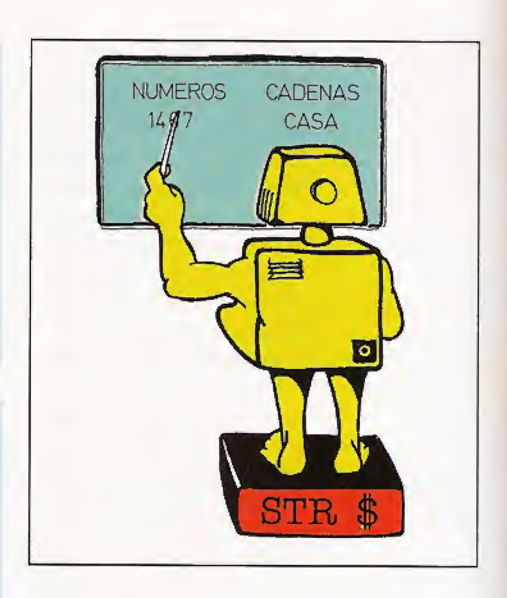
Ejemplos: STR$(124)   STR$(C)
STR$(A)   STR$(2458)

VAL

Función: Convierte una cadena en un número.

Formato: VAL (<cadena>)
<cadena>: cadena de caracteres que se desea convertir en un número.

Ejemplos: VAL(A$)   VAL(T$)
VAL("128")   VAL("2578")



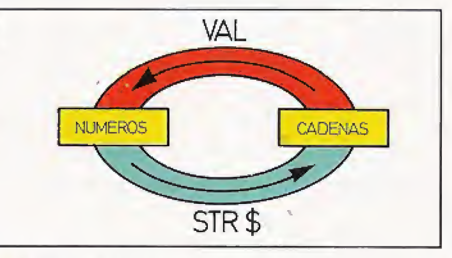
meros aparecen en pantalla precedidos por uno o dos espacios en blanco; espacios que son empleados para la representación del signo y que suelen aparecer siempre, aunque el signo no sea visualizado.

A continuación figura un ejemplo que pone de manifiesto la analogía entre ambas representaciones:

```
10 PRINT "128"
20 PRINT 128
RUN <CR>
128
 128
█
```

Obsérvese que la segunda presentación del número 128 está desplazada con respecto a la posición inicial de la primera. Ello se debe al motivo anteriormente expuesto.

Los usuarios del ZX-SPECTRUM, notarán que su ordenador no reserva ese espacio para el signo, actuando, en este caso, de forma distinta a la mayor parte de equipos dotados de un traductor BASIC.



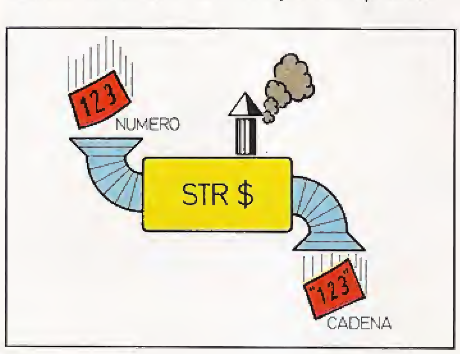
Las funciones STR$ y VAL constituyen un puente que permite la transformación de datos numéricos en caracteres, y viceversa.

La diferencia entre datos numéricos y alfanuméricos no se limita a una cuestión tan trivial como ésta. Se refleja fundamentalmente en el tipo de operaciones que es posible realizar con ellos. Por ejemplo, no es lo mismo realizar una suma de números que una suma de cadenas de caracteres. El resultado de ambas operaciones es bien distinto, como revela el siguiente ejemplo:

```
10 PRINT 128+4
20 PRINT "128" + "4"
RUN <CR>
 132
1284
█
```

En él se observa que la operación "+" realiza una función distinta dependiendo del tipo de datos. Si los datos son numéricos, el resultado será la suma aritmética. Por el contrario, si se trata de cadenas de caracteres, el resultado será su concatenación.

La conversión de datos numéricos en cadenas de caracteres resultará útil para manipular datos numéricos a modo de cadenas de caracteres. En efecto, hay que recordar que existen operaciones fácilmente realizables con cadenas, pero que, sin embargo, son difíciles de realizar con números. Por ejemplo, para extraer la cifra que en un determinado número corresponde a las centenas, se-

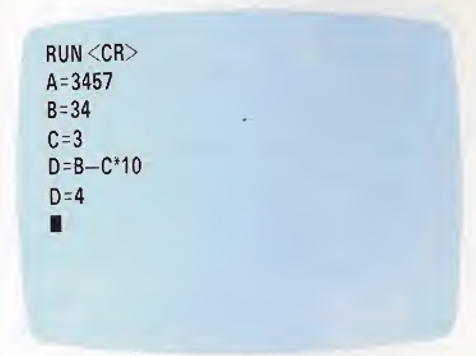


La función STR$ da como resultado una cadena cuyos caracteres coinciden con las cifras del número incluido en su argumento.

ría necesario recurrir a un programa semejante al que sigue:

```
 10 A=3457
 20 PRINT "A= ";A
 30 B=INT(A/1000)
 40 PRINT "B= ";B
 50 CINT(A/1000)
 60 PRINT "C= ";C
 70 D=B—C*10
 80 PRINT "D=B—C*10"
 90 PRINT "D= ";D
100 END
```

Comprobemos su eficacia:



Esta rutina procede a dividir el dato base (A) por 100 y por 1000 sucesivamente. Con la primera división se coloca la cifra de centenas a extraer (en este caso 4) en la posición de las unidades. La siguiente división obtiene las cifras que quedan a la izquierda de la elegida. Por último, la operación de la línea 70 aisla el dígito buscado, almacenándolo en la variable D.

La misma operación se puede realizar con mayor facilidad e inmediatez si el dato inicial es una cadena de caracteres. La función MID$, comentada en un capítulo anterior, permitirá la extracción del carácter deseado:

```
10 A=3457
20 A$=STR$(A)
30 C$=MID$(A$,LEN(A$)—2,1)
40 PRINT C$
RUN <CR>
4
█
```

Ahora se consigue la extracción con una sola línea de programa, la 30. Para ello es necesario almacenar el dato inicial en una variable de cadena (línea 20). De esta forma el dato numérico queda transformado en una cadena de caracteres.

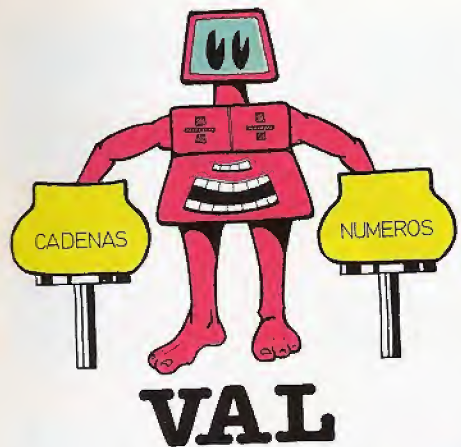
Como se ha podido comprobar, la función STR$ permite almacenar un número en una variable de cadena. Este paso hace posible realizar algunos tipos de operaciones que resultan difíciles, y a veces imposibles, con números.

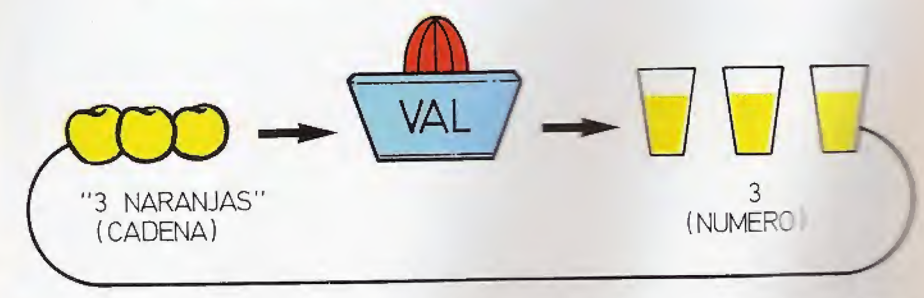
También es posible realizar la conversión inversa. Es decir, el paso de cadena de caracteres a dato numérico. Para ello hay que emplear la función VAL cuya especialidad es obtener el valor numérico asociado a una cadena de caracteres. Desde luego, es necesario que la cadena inicial esté constituida por cifras. Su formato es el siguiente:

VAL (<cadena>)

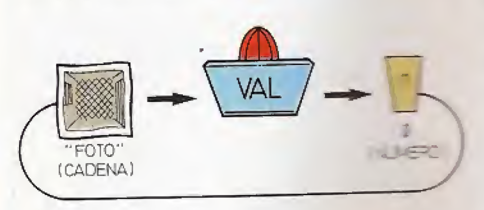
El argumento <cadena> representa al dato alfanumérico que se desea convertir en un número. He aquí algunos ejemplos:

```
VAL (A$)
VAL ("272")
VAL ("127")
VAL (T$)
```





La función VAL obtiene el valor numérico asociado a la cadena de caracteres que constituye su argumento. Concretamente, transforma en dato numérico los primeros caracteres de la cadena que coincidan con cifras.



Si la cadena evaluada no empieza por un carácter numérico, la función VAL genera como resultado el valor cero.

Su actuación se reduce a convertir los caracteres de la cadena que correspondan a números en sus respectivos valores numéricos. En el caso de que la cadena no esté compuesta únicamente por cifras, se realizará la conversión sólo hasta el primer carácter no numérico. Si se da la circunstancia de que el primer carácter de la cadena no es un número, el valor devuelto por esta función será cero.

En el BASIC de Sinclair (ZX-SPECTRUM y ZX-81), la función STR$ presenta una singularidad: si al evaluar una cadena de caracteres se encuentran caracteres no numéricos, éstos se tomarán como nombres de variables; se buscará su valor correspondiente y se operará con ellos. En otras palabras, la cadena es interpretada cual si se tratara de una fórmula.

Generación de cadenas

En ocasiones es necesario crear cadenas de caracteres de una determina-

SPACE$

Función: Genera una cadena de caracteres, compuesta por la repetición de espacios en blanco un determinado número de veces.

Formato: SPACE$ (<número>)
<número>: número de espacios en blanco que deben conformar la cadena.

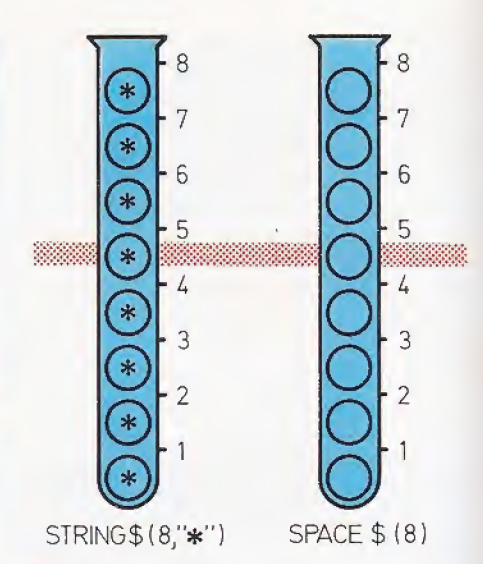
Ejemplos:
```
SPACE$(7)          SPACE$(12)
SPACE$(N)          SPACE$(S)
```

STRING$

Función: Genera una cadena de caracteres formada por la repetición, un determinado número de veces, del carácter que se especifique.

Formato: STRING$ (<num1>,<num2>
ó
STRING$(<núm1>,<cadena>)
<num1>: número de caracteres que han de formar la cadena.
<num2>: código ASCII de carácter a utilizar.
<cadena>: el primer carácter de esta cadena será el que constituya la nueva cadena.

Ejemplos:
```
STRING$(7,"CASA")          STRING$(N,P)
STRING$(12,56)             STRING(N,A$)
```



STRING$ construye una cadena integrada por la repetición del carácter que indique el programador, mientras que SPACE$ sintetiza la cadena resultante a base de espacios en blanco.

da longitud y que estén formadas únicamente por espacios en blanco. Quizá el lector se pregunte cuál es la utilidad práctica de semejante tipo de cadena. Suponga, por ejemplo, que es necesario disponer de una cadena de longitud específica. Dicha cadena estará formada por una serie de caracteres que constituyen realmente el dato con el que se ha de trabajar. Nada garantiza que el número de caracteres vaya a ser precisamente el estipulado. Lo más normal es que el espacio destinado a contener la información sea grande, y que sobre sitio para contener la cadena. Por ello, es necesario llenar el espacio que sobra con caracteres en blanco. Esta puede ser una alternativa útil, por ejemplo, a la hora de almacenar información en un archivo o presentarla en pantalla (recuerde que no todos los ordenadores poseen la socorrida instrucción PRINT AT).

Una solución al problema planteado consiste en utilizar la siguiente rutina, que no por simple deja de ser menos interesante y efectiva.

```
 5 INPUT B
10 LET B$=" "
20 FOR I=1 TO B
30 LET B$=B$+" "
40 NEXT I
```

La ejecución de esta rutina crea una cadena B$ formada por un número B de espacios en blanco. Todo su secreto reside en añadir blancos a una cadena vacía. De todas formas, este mismo resultado se puede obtener mediante el empleo de la función SPACE$. Siempre y cuando éste forme parte del repertorio del BASIC del ordenador utilizado. Su formato es el siguiente:

SPACE$ (<número>)

La zona <número> estará ocupada por un número natural, cuya misión es indicar al ordenador el número de espacios en blanco que van a conformar la cadena.

```
SPACE$ (6)
SPACE$ (N)
```

Volviendo al ejemplo anterior, hubiera bastado con la siguiente línea de programa para conseguir el mismo resultado:

```
LET B$=SPACE$ (B)
```

La función SPACE$ puede utilizarse en compañía de otras funciones para el tratamiento de cadenas de caracteres. La siguiente rutina hará que cualquier cadena introducida se imprima en pantalla, ocupando diez espacios delimitados por el símbolo "/".

```
100 INPUT B$
110 LET A$=SPACE$ (10—LEN (B$))
120 LET B$=B$+A$
130 PRINT "/";B$;"/"
```

En la línea 110 se hace uso de LEN para calcular los espacios que le faltan a B$ para llegar a los 10 caracteres de



STRING$ es una función BASIC que permite construir cadenas de una determinada longitud a partir de la repetición de un determinado carácter.

longitud. El cálculo constituye el argumento de SPACE$, logrando así una cadena (A$) con el número adecuado de blancos. La línea 120 es la encargada de ensamblar ambas cadenas, para formar una cadena final de 10 caracteres de longitud. Por último, en la línea 130 se imprime la cadena obtenida, flanqueada por los símbolos "/". En la práctica, la mencionada rutina puede simplificarse de la siguiente forma:

```
100 INPUT B$
110 PRINT "/";B$+SPACE$(10—LEN(B$));"/"
```

Ahora, todas las operaciones se realizan en la línea 110.

La posibilidad de formar cadenas de una determinada longitud, integradas por la repetición un solo carácter, no se limita a los espacios en blanco. También es posible realizar la misma operación con otros caracteres. Para ello el BASIC dispone de la función STRING$, cuya formulación es la que sigué:

STRING$ (<n1>,<n2>) ó
STRING$ (<n1>,<cadena>)

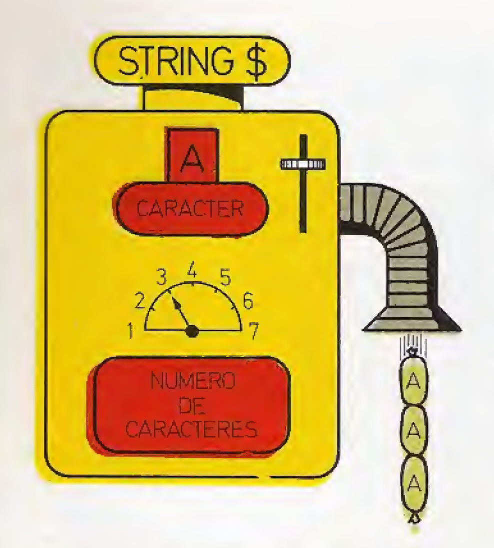
Donde <n1> es un número natural que indica la longitud de la cadena y <n2> corresponde al código ASCII del carácter a repetir. En el formato alternativo, <cadena> representa a una cadena de caracteres, cuyo carácter inicial es el que desea repetirse <n1> veces.

Como se observa, esta función admite dos formulaciones alternativas. En ambos casos se crea una cadena formada por la repetición de un carácter. El carácter a repetir coincide, o bien con el que se indica mediante su código ASCII, o con el primero de la cadena que se especifica. El número de veces que se repite el carácte en cuestión viene da do por el valor del parámetro <n1>.

El mismo resultado del primer formato puede lograrse con la siguiente rutina:

```
 5 INPUT "VECES";I
 8 INPUT "CODIGO ASCII";K
10 LET B$=""
20 FOR J=1 TO I
30 LET B$=B$+CHR$(K)
40 NEXT J
50 PRINT B$
```

En ella, la variable I contiene el número de caracteres que ha de contener la cadena. Esta variable se utiliza para

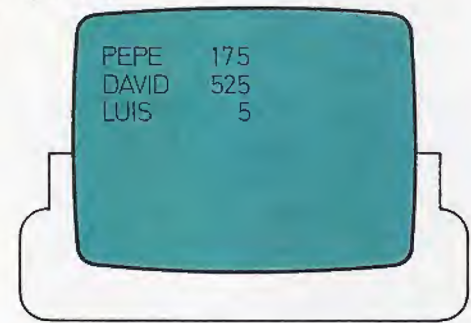


La puesta en actividad de la función STRING$ exige dos datos: el carácter que hay que repetir y el numérico de caracteres que deben constituir la cadena resultante.

marcar el límite del bucle FOR. A su vez, K almacena el código ASCII del carácter adecuado. La línea 30, la cual se repite el número de veces estipulado por la variable I, añade el referido carácter a la cadena final cuantas veces sea preciso.

El segundo formato sería algo más complicado de programar; aunque el problema tampoco es excesivo. Veamos una posible solución:

```
 5 INPUT "VECES";I
 8 INPUT "CADENA";X$
10 LET B$=""
20 LET K$=LEFT$(X$,1)
30 FOR J=1 TO I
40 LET B$=B$+K$
50 NEXT J
60 PRINT BS
```



Una de las aplicaciones de las funciones STRING$ y SPACE$ es la confección de formatos de pantalla.

La diferencia con la primera rutina reside en la línea 20. Aquí es necesario extraer el primer carácter de la cadena introducida en X$.

Un uso elemental, aunque práctico, de la función que nos ocupa, puede ser la simple sustitución de la instrucción que sigue:

```
PRINT "********************************"
```

por otra bastante más elegante:

```
PRINT STRING$(32,"*")
```

Con ella se evita la necesidad de escribir 32 veces el asterisco; evitando, además, la posibilidad de cometer algún error a la hora de contar los caracteres a escribir.

Búsqueda de subconjuntos

En lo relativo al manejo de subcadenas o porciones de una cadena de caracteres, se han estudiado en anteriores capítulos las funciones adecuadas para su extracción: LEFT$, RIGHT$ y MID$. Queda aún otra operación importante que es posible realizar con subcadenas: detectar si una determinada cadena o subcadena está incluida en otra. Esta es una actividad encomendable a la función INSTR, cuyo formato es el siguiente:

INSTR (<n1>,<cadena1>,<cadena2>)

En donde <n1> es un número natural, y <cadena 1> y <cadena 2> son dos cadenas de caracteres. Los siguientes son ejemplos de uso correcto de esta función.

```
INSTR (A$,H$)
INSTR (5,D$,G$)
```

Su funcionamiento es el siguiente: la instrucción examinará la <cadena 1> y comprobará si contiene a la <cadena 2>. En el caso de que ello suceda, devolverá como resultado un número. Este corresponderá a la posición que ocupa el primer carácter de la <cadena 2> dentro de la <cadena 1> bajo examen. Si no se encuentra dicha subcadena, el resultado coincidirá con un cero. Idéntica respuesta se obtiene si la cadena en la que se realiza la búsqueda (<cadena 1>) es nula.

INSTR

Función: Busca una secuencia de caracteres dentro de otra, devolviendo como resultado la posición del primer carácter de la secuencia dentro de la cadena.

Formato: INSTR(I,A$,B$)

I: posición a partir de la cual deseamos que se realice la búsqueda.
A$: cadena en la que deseamos buscar la repetición.
B$: secuencia de caracteres que deseamos saber si están contenidos dentro de la cadena A$.

Ejemplos: INSTR(2,X$,P$)
INSTR(5,"CASAS","AS")
INSTR(C$,F$)
INSTR(3,D$,G$)

La especialidad de la función INSTR reside en detectar la presencia de una determinada cadena o subcadena dentro de otra.

```
10 LET A$="COMIENZO"
20 LET B$="MI"
30 PRINT INSTR (A$,B$)
RUN <CR>
3
█
```

En el ejemplo se observa que INSTR devuelve la posición a partir de la cual se encuentra la subcadena «MI» dentro de la cadena inicial «COMIENZO»; en este caso se trata de la posición 3. Este dato puede ser utilizado posteriormente en el argumento de una función MID$ o LEFT$, para trocear la cadena por el punto detectado.

Si se desea iniciar la búsqueda a partir de un carácter distinto del primero, se puede especificar dicho carácter en el argumento de INSTR. Par ello se indicará la posición a partir de la cual debe empezar el proceso de búsqueda: <n 1>.

```
10 LET A$="LILIPUTIENSE"
20 LET B$="LI"
30 PRINT INSTR (A$,B$)
40 PRINT INSTR (3,A$,B$)
RUN <RUN>
1
3
█
```

El ejemplo revela la diferencia entre lo que sucede cuando se especifica o no el carácter de partida para el proceso de búsqueda. La instrucción de la línea 30 localiza la subcadena «LI» en el primer carácter; la respuesta que se obtiene coincide, precisamente, con la posición de dicho carácter: 1. Sin embargo, en el segundo caso se prescinde de esta primera subcadena, puesto que no son examinados los dos primeros caracteres de la cadena de origen. Al respecto, hay que tener en cuenta que si el número que se especifica como comienzo de la búsqueda es mayor que la longitud de la cadena, el resultado de la función INSTR será cero.

Definición de variables de cadena

En un capítulo previo se estudió la forma en la que se podían definir ciertas variables numéricas, especificando su formato. Ello obviaba la necesidad de especificarlo mediante un carácter especial, colocado al final del nombre de la variable (!,% o *). El BASIC ofrece la posibilidad de realizar la misma operación con variables alfanuméricas. En este caso la definición ha de realizarse mediante el comando DEFSTR. Su ejecución evitará la necesidad de incluir el carácter «$» al final de los nombres de variables alfanuméricas previamente definidos. Su formulación presenta el mismo aspecto que la propia de sus homólogos numéricos.

DEFSTR

Función: Define como cadena de caracteres las variables que comiencen con determinadas letras.

Formato: DEFSTR <rango de caracteres>

Ejemplo: DEFSTR A—D,S,U,X
DEFSTR U
DEFSTR A—F

TABLA DE CONVERSION						
Ordenador	STR$	VAL	SPACE$	STRING$	INSTR	DEFSTR
	STR$ (<número>)	VAL (<cadena>)	SPACE$ (<número>)	STRING$ (<num1 >,<num2>)	INSTR ([I,]A$,B$	DEFSTR
AMSTRAD	STR$(<número>)	VAL(<cadena>)	SPACE$(<número>)	STRING$(<num1>,<num2>)	INSTR([I,]A$,B$	DEFSTR
APPLE II (APPLESOFT)	STR$(<número>)	VAL(<cadena>)	—	—	—	—
APRICOT (M-BASIC)	STR$(<número>)	VAL(<cadena>)	SPACE$(<número>)	STRING$(<num1>,<num2>)	INSTR([I,]A$,B$	DEFSTR
ATARI	STR$(<número>)	VAL(<cadena>)	—	—	—	—
CBM 64	STR$(<número>)	VAL(<cadena>)	—	—	—	—
DRAGON	STR$(<número>)	VAL(<cadena>)	—	STRING$(<num1>,<num2>)	INSTR([I,]A$,B$	DEFSTR
EQUIPOS MSX	STR$(<número>)	VAL(<cadena>)	SPACE$(<número>)	STRING$(<num1>,<num2>)	INSTR([I,]A$,B$	DEFSTR
HP-150	STR$(<número>)	VAL(<cadena>)	SPACE$(<número>)	STRING$(<num1>,<num2>)	—	DEFSTR
IBM PC	STR$(<número>)	VAL(<cadena>)	SPACE$(<número>)	STRING$(<num1>,<num2>)	INSTR([I,]A$,B$	DEFSTR
MPF	STR$(<número>)	VAL(<cadena>)	SPACE$(<número>)	—	—	—
NCR DM-V(MS-BASIC)	STR$(<número>)	VAL(<cadena>)	SPACE$(<número>)	STRING$(<num1>,<num2>)	INSTR([I,]A$,B$	DEFSTR
NEW BRAIN	STR$(<número>)	VAL(<cadena>)	—	—	INSTR(A$,B$[,I])(1)	—
ORIC	STR$(<número>)	VAL(<cadena>)	—	—	—	—
SHARP MZ-700 (MZ-BASIC)	STR$(<número>)	VAL(<cadena>)	—	—	—	—
SINCLAIR QL	—	—	—	FILL(<cad>, <num>) (2)	A$ INSTR B$ (3)	—
SPECTRAVIDEO	STR$(<número>)	VAL(<cadena>)	SPACE$(<número>)	FILL(<cad>,<num>) (2)	A$ INSTR B$ (3)	DEFSTR
ZX-SPECTRUM	STR$(<número>)	VAL(<cadena>)	—	—	—	—

1. **Función INSTR en el NEW BRAIN**
   Busca la cadena B$ en A$ y devuelve la posición del primer carácter de B$ en A$. Si esta cadena no es localizada, devuelve el valor 0.
   Si se especifica el número 1, la búsqueda comenzará a partir del carácter situado en la posición 1.

Formato: *INSTR (A$, B$[, 1)*
A$, B$: cadena de caracteres
I: número entero
Ejemplos: INSTR ("PATATA","TA",2)
INSTR ("CAER","ER")

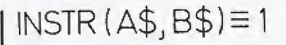
2. **Función FILL$ del QL, equivalente a STRING$**
   Crea una cadena de <num> caracteres, formada por la repetición de la cadena de caracteres especificada en <cad>.

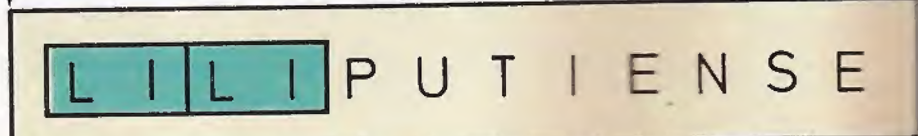
Formato: *FILL$ (<cad>,<num>)*
<cad>: cadena de caracteres
<num>: número entero
Ejemplos: PRINT FILL$("OA",5)<CR>
OAOAO
PRINT FILL$("*",3)<CR>
***

3. **Función INSTR en superBasic QL**
   Busca la cadena A$ en B$

Formato: *A$INSTR B$*
Ejemplo: "CA" INSTR "PETACA"

Actuación de INSTR con diversas formulaciones. La cadena que hay que buscar coincide en el ejemplo con los caracteres «LI».





DEFSTR <rango de caracteres>

En la zona <rango de caracteres> se especifican las iniciales de los nombres de variables a definir, separados por comas. Si los caracteres son consecutivos, se puede citar únicamente el primer y último caracteres, separados por un guión. A continuación se incluyen algunos ejemplos relativos al uso de este comando.

DEFSTR A,M—N

Indica que las variables que comiencen por A, M y N serán de cadena de caracteres.

DEFSTR A,C,M—P

En este caso, las variables que comiencen por A, C, M, N, O y P quedan definidas como alfanuméricas.



Sistemas de numeración

Desde los tiempos más remotos, el hombre ha utilizado sistemas de numeración para cuantificar objetos y acontecimientos. Enumerar y llevar la cuenta de todo tipo de objetos ha sido una necesidad atemporal, necesidad que dio pie al desarrollo de sistemas de numeración que le permitieran conocer sus pertenencias, sus deudas o su edad. Entre los diversos sistemas inventados, el más extendido es el decimal. La razón es muy simple: poseemos diez dedos en las manos.

Seguramente, el primer contable de la historia utilizó sus dedos. Aún hoy en día estos apéndices son considerados como la «calculadora» más elemental. El principal defecto de este método de cálculo reside en el hecho de que sólo se puede llegar hasta diez. Para cifras mayores es necesario tomar una referencia externa.

Suponga que se desea contar una serie de objetos. La cuenta empieza extendiendo un dedo por cada objeto contabilizado. Al llegar al décimo objeto se terminan los dedos. Este problema puede solventarse buscando a alguien que cuente las unidades de diez dedos acumuladas. Al final de la cuenta se verifican las decenas y los dedos sueltos, con lo que se obtiene la cantidad total.

Cuando el segundo contador termina sus dedos, es necesario llamar a un tercero. Este último contará las centenas o decenas de decenas. Y así sucesivamente.

Este es, justamente, el método que se utiliza para escribir cifras. La cantidad 23 indica 2 decenas y 3 «dedos sueltos» o unidades. Cuando se acumulan más de 9 decenas se hace uso de un tercer número, revelador de las centenas.

El sistema de numeración expuesto se denomina *decimal* o de base diez. Ello se debe a que se utiliza un nuevo dígito para cada *diez* unidades precedentes.

En todo caso, el sistema decimal no es el único sistema de numeración posible. De hecho, muchos pueblos de la antigüedad adoptaron otros sistemas. Sin embargo, bien es cierto que actualmente, el sistema de numeración estándar es el decimal. El ordenador no tiene dedos. Maneja los números bajo el aspecto de impulsos eléctricos; y de forma muy simple: tan sólo detecta la presencia o ausencia de tensión. La manipulación de los datos se efectúa por medio de interruptores: si el interruptor está cerrado pasa electricidad, si está abierto no. Ello equivale a datos del tipo SI o NO (pasa corriente).

A la hora de contar, el ordenador sólo puede adoptar dos estados distintos, lo que significa que puede contar hasta dos directamente. Para cantidades mayores, necesita un nuevo circuito que cuente unidades de dos o «pares». El siguienbte circuito contaría pares de pares, esto es unidades de cuatro. Y, en definitiva, aumentando convenientemente el número de circuitos se puede llegar a cualquier cantidad.

El sistema de numeración empleado por los ordenadores se basa, pues, en la detección de dos estados perfectamente diferenciados. Es un sistema de base dos o *binario*. La forma de representar las cantidades en este sistema es análoga a la utilizada para el decimal. El primer dígito de la derecha indica las unidades; cuando se llega a dos se añade un nuevo dígito a la izquierda, etc.

Según esto, para representar a la unidad bastaría con poner el dígito «1». Sin embargo, para dos unidades se ha de utilizar otro dígito adicional. La representación de dos, en binario será «10»: el primer dígito (1) indica un par y el segundo (0) cero unidades más. Análogamente tres se escribe «11», cuatro «100», etc.

El sistema binario sólo exige dos símbolos para elaborar representaciones «0» y «1». Análogamente, el sistema decimal usa diez: del 0 al 9. Otro sistema empleará el número de símbolos indicados por su base. Así, el sistema de numeración de base seis utilizará seis dígitos: del 0 al 5.

A la hora de trabajar con ordenadores resulta útil conocer el sistema binario. Con ello será posible comprender y asimilar mejor los procesos que tienen lugar en su interior.

El sistema binario tiene, no obstante, claros inconvenientes. El primero es la novedad. Estamos acostumbrados al decimal y cualquier otro sistema resulta complicado de manejar, al menos inicialmente. La segunda pega del sistema binario es la extensión de su representación. Para escribir 8 unidades hacen falta cuatro dígitos («1000»), y para representar 1000 unidades serán necesarios 9 dígitos. Este inconveniente hace tedioso el manejo en binario de grandes cantidades. La solución al problema consiste en apoyarse en sistemas de numeración alternativos. El sistema ideal sería uno que fuera sencillo de convertir al binario y que, además, su base fuera cercana a la decimal. Para cumplir la primera premisa, es indudable que su base ha de ser una potencia de 2. Las bases que satisfacen ambos requisitos son 4, 8, 16... Las dos más recurridas son la 8 y 16, que corresponden a los sitemas octal y hexadecimal respectivamente.

El sistema octal emplea ocho símbolos, coincidentes con las cifras decimales del 0 al 7. El paso de octal a binario y viceversa es realmente sencillo. Para ello basta conocer el formato binario de los 8 primeros números en el sistema octal. A su vez, el paso de binario a octal se realiza agrupando los dígitos en conjuntos de tres y traduciéndolos bloque a bloque. Por ejemplo, el número binario 110100111 equivale al octal 647 (110=6, 100=4, 111=7). Para realizar la operación inversa, se ha de «traducir» cada dígito octal por tres binarios, rellenado con tantos ceros a la izquierda como sea necesario. Así, por ejemplo, el número octal 121 corresponde al binario 1010001 (1=1, 2=010, 1=001).

El sistema de numeración hexadecimal necesita 16 símbolos para la representación de sus dígitos. Esto significa que empleando las cifras del 0 al 9 faltan aún seis símbolos para completar las exigencias. El método habitual es utilizar letras para los restantes; las más comúnmente empleadas son las seis primeras mayúsculas: A, B, C, D, E y F. El paso de binario a hexadecimal es similar al que se aplica con el sistema octal, sin más que agrupar esta vez los dígitos binarios de cuatro en cuatro.

Decimal	Binario	Hexadecimal	Octal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17
16	10000	10	20
17	10001	11	21
18	10010	12	22
19	10011	13	23
20	10100	14	24

Subrutinas

Entre GOSUB y RETURN



En la terminología informática, una *rutina* es un bloque de instrucciones destinado a cumplir una misión específica dentro de un programa. La propia denominación, «rutina», sugiere tal definición. Y es que, realmente, el programa no es más que un encadenamiento de rutinas que se ejecutan unas tras otra.

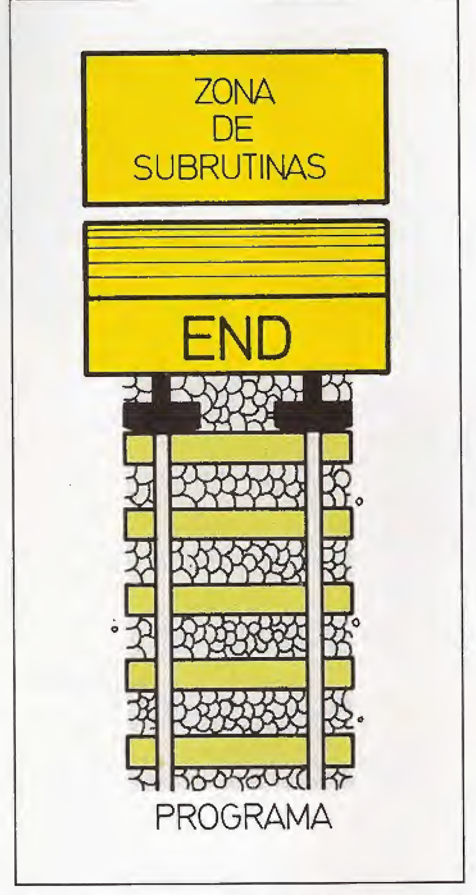
Por pura rutina

En la estructura de cualquier programa cabe aún otra subdivisión, además de la establecida a partir de los módulos funcionales o rutinas. Aquí entra en liza la *subrutina*. De nuevo, una subrutina es un conjunto de instrucciones que forman un pequeño programa, anexo al programa principal, situado generalmente al final de éste (más adelante se verá el motivo).

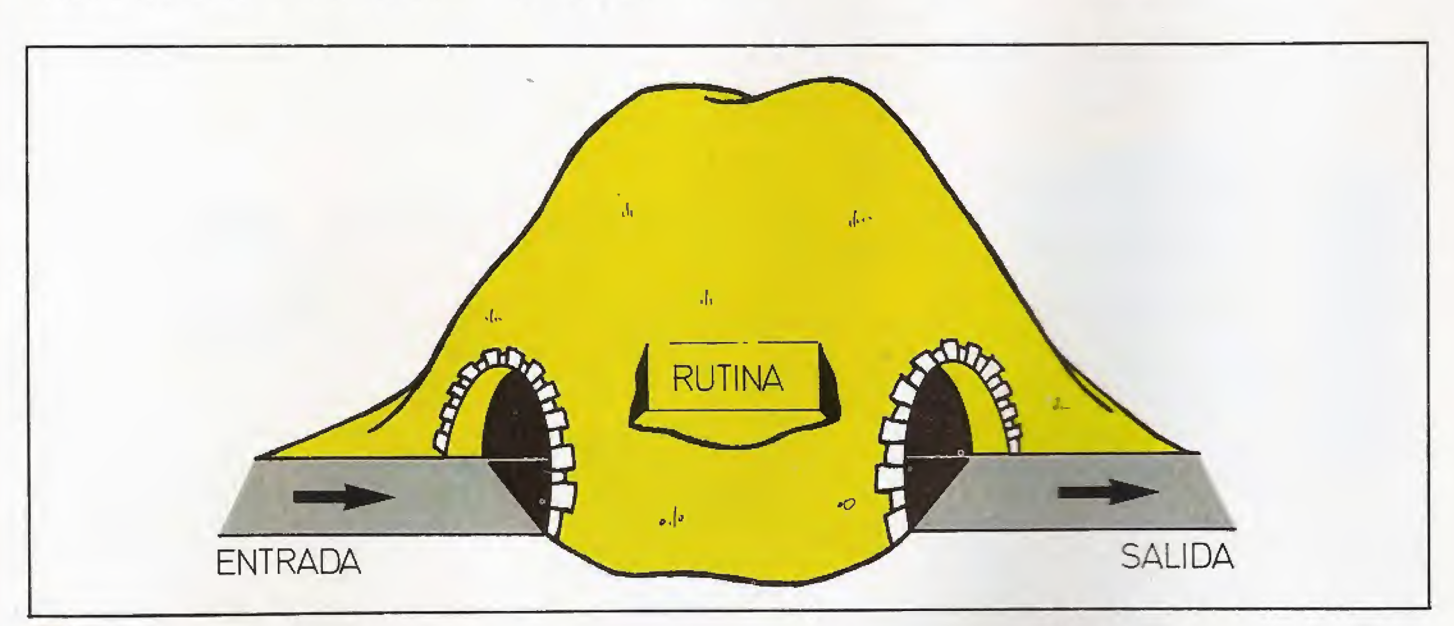
Las subrutinas —de menor entidad y volumen que las rutinas o módulos del programa— estarán constituidas por una serie de instrucciones, cuya ejecución se repetirá varias veces durante el desarrollo del programa. De inmediato saltan a la vista las primeras ventajas de las subrutinas: obvian la necesidad de repetir varias veces, y dentro del mismo programa, un mismo conjunto de instrucciones. Con el empleo de subrutinas sólo será necesario escribir las zonas coincidentes una sola vez, y acceder a ellas cuantas veces haga falta a lo largo del programa. Por lo tanto, es obvio que el uso de subrutinas contribuirá a rentabilizar la memoria del ordenador, economizando el espacio que ocuparían los bloques de instrucciones reemplazados por cada rutina.

En el sencillo ejemplo que acompaña a estas líneas, se puede observar cómo la repetición exhaustiva de las instrucciones que sirven para ingresar un número por el teclado, convierten al programa en monótono y repetitivo. Tal reiteración puede desaparecer drásticamente apelando a una subrutina, como se verá más adelante.

```
10 REM EJERCICIO REPETITIVO
20 PRINT "INTRODUZCA UN NUMERO ENTRE 1 Y 100"
30 INPUT PRIMNUM
```



La instrucción END (fin del programa) constituye la barrera infranqueable en la que termina la secuencia de un programa BASIC. Tras ella pueden situarse las subrutinas, que quedan así protegidas de cualquier ejecución accidental.



Una rutina es un bloque de instrucciones destinado a cumplir una misión específica dentro del programa. Normalmente, la rutina sólo será ejecutada en una ocasión durante el desarrollo del programa.

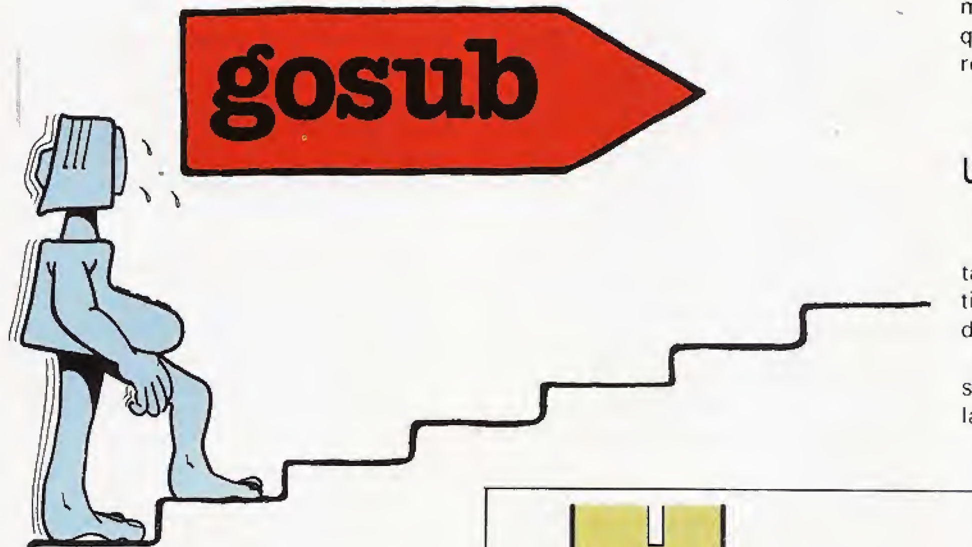
```
 40 IF PRIMNUM<1 OR PRIMNUM>100 THEN GOTO 20
 50 PRINT "INTRODUZCA UN NUMERO ENTRE 1 Y 100"
 60 INPUT SEGUNUM
 70 IF SEGUNUM<1 OR SEGUNUM>100 THEN GOTO 50
 80 PRINT "INTRODUZCA UN NUMERO ENTRE 1 Y 100"
 90 INPUT TERCNUM
100 IF TERCNUM<1 OR TERCNUM>100 THEN GOTO 80
110 LET MAYOR=PRIMNUM
120 IF SEGUNUM>MAYOR THEN LET MAYOR=SEGUNUM
130 IF TERCNUM>MAYOR THEN LET MAYOR=TERCNUM
140 PRINT "EL NUMERO MAYOR ES:";MAYOR
150 END
```

GOSUB

Transfiere el control del programa a la subrutina cuyo número de línea inicial se especifica.

Formato: <NI> GOSUB [<nl> <expresión>]

Ejemplos: 10 GOSUB 1000
50 GOSUB A+100



La ejecución del programa dará el siguiente resultado en la pantalla:

```
INTRODUZCA UN NUMERO ENTRE 1 Y 100
?45
INTRODUZCA UN NUMERO ENTRE 1 Y 100
?83
INTRODUZCA UN NUMERO ENTRE 1 Y 100

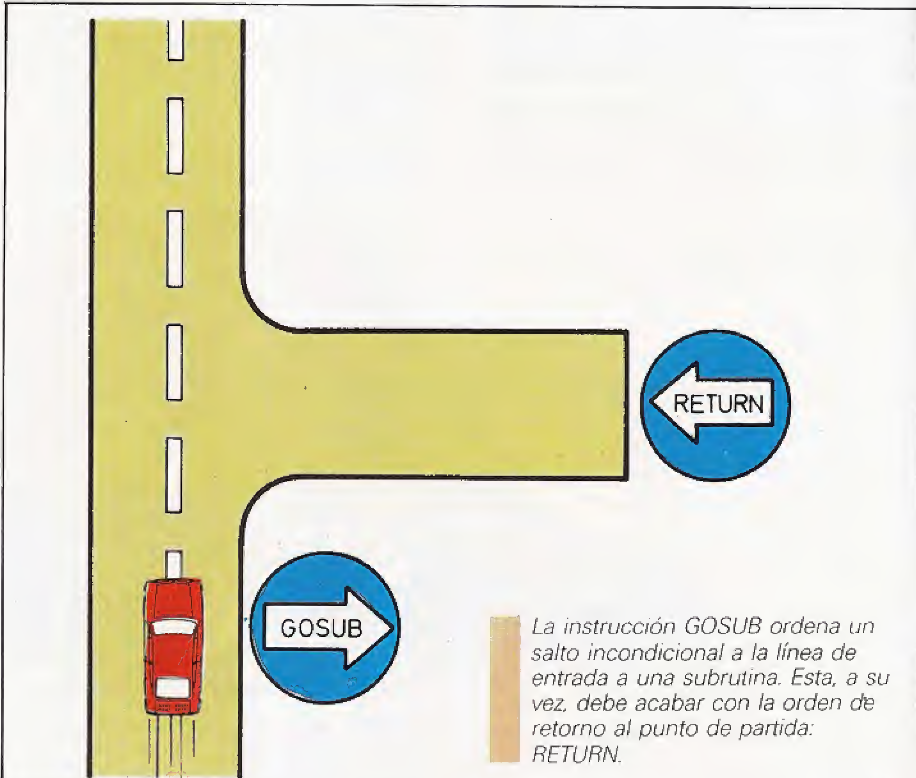
?27
EL NUMERO MAYOR ES: 83
█
```

El programa pide la introducción sucesiva de tres números a través del teclado, dando como resultado el mayor de los tres. Se trata de un ejemplo sencillo aunque elocuente e ilustrativo de la frecuente necesidad de repetir las mismas instrucciones dentro de un mismo programa. En el ejemplo, la zona que solicita la introducción de un número aparece en tres ocasiones.

Un viaje al exterior

Llegados a este punto, cabe preguntarse cómo hay que construir las subrutinas y de qué forma se pueden utilizar dentro de un programa.

Para que sea posible acceder a una subrutina será preciso desviar hacia ella la secuencia de ejecución del programa.



La instrucción GOSUB ordena un salto incondicional a la línea de entrada a una subrutina. Esta, a su vez, debe acabar con la orden de retorno al punto de partida: RETURN.

on gosub

Una vez ejecutado todo el conjunto de instrucciones de la subrutina, habrá que regresar al punto en el que se abandonó la ejecución del programa principal al producirse la llamada a la subrutina. ¡Un viaje de ida y vuelta!

El lenguaje BASIC cuenta con un comando especializado en este cometido; GOSUB (GO SUBrutine). Este comando se introducirá en cualquier punto del programa en el que se desee acceder a una subrutina.

El comando GOSUB debe ir precedido, como es habitual, por su correspondiente número de línea. Tras la palabra clave GOSUB se indicará el número de línea a partir del cual se encuentra situada la subrutina a la que se quiere acceder. Por consiguiente, el formato general de una instrucción GOSUB coincidirá con el siguiente:

<Núm. de línea> GOSUB <Núm. de línea de la subrutina>

Por ejemplo:

```
100 GOSUB 2000
```

Cuando la ejecución del programa llegue a la línea que contiene la instrucción GOSUB, la secuencia saltará a la nueva línea cuyo número se indica. La ejecución continuará normalmente a partir de este punto del programa, coincidente con la entrada a la subrutina.

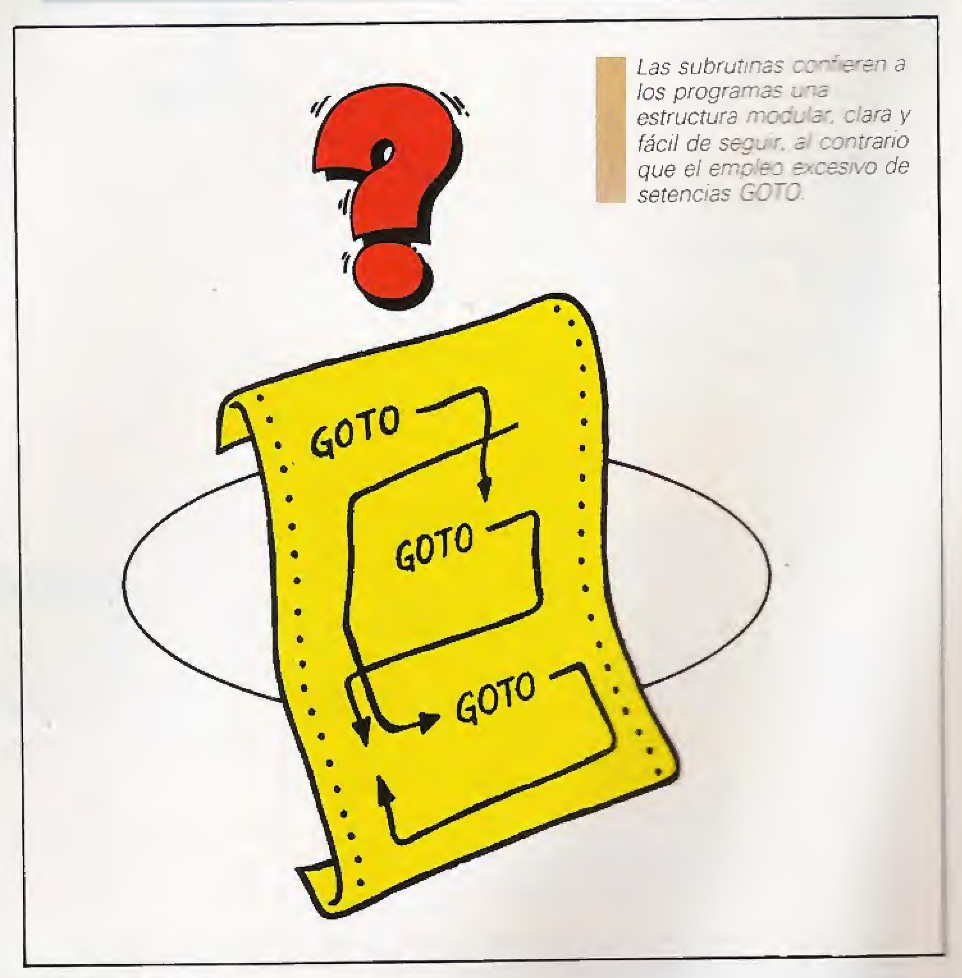
Al llegar a una instrucción GOSUB, el microprocesador memoriza la dirección que corresponde a la instrucción que sigue a la de llamada a subrutina. De esta forma, conocerá con exactitud la dirección a la que debe de retornar.

ON/GOSUB

Transfiere el control a un subrutina de forma condicional y selectiva. El salto se producirá tras evaluar la expresión que ocupa la zona ON; concretamente, a la subrutina cuyo número de línea ocupa, en el argumento de GOSUB, la posición señalada por el resultado de la expresión.

Formato: <Nl> ON <exp.> GOSUB <nl1>[, <nl2>,...,<nlk>]

Ejemplos:
```
10 ON X GOSUB 1000, 2000
80 ON X+10-B GOSUB 100,200,300,400
```



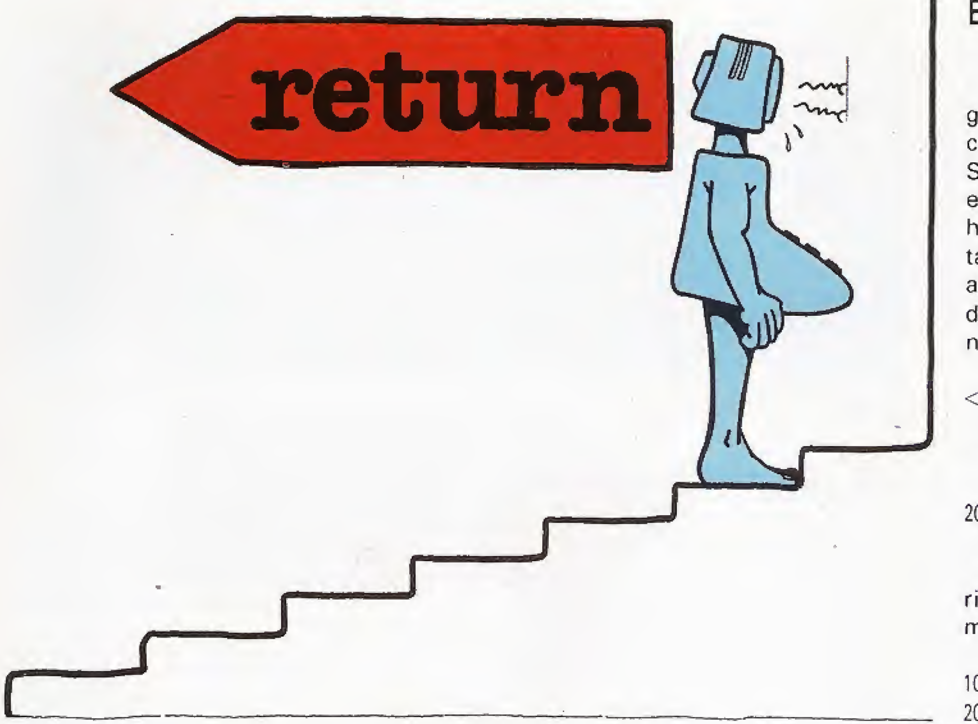
Las subrutinas confieren a los programas una estructura modular, clara y fácil de seguir, al contrario que el empleo excesivo de setencias GOTO.

RETURN

Señala el final de una subrutina; devuelve elcontrol del programa a la siguiente instrucción a la que efectuó la llamada.

Formato: <NL> RETURN

Ejemplo: 100 RETURN



El viaje de vuelta

La orden de regreso al punto de origen se introduce como última instrucción de la subrutina. En el lenguaje BASIC la orden al efecto es RETURN. Al ejecutarla, el ordenador repara en que ha llegado al final de la subrutina, y por tanto que se debe «retornar» (RETURN) al punto de llamada. Su formato se reduce a la palabra clave precedida por el número de línea que le corresponda:

<Núm. de línea> RETURN

Por ejemplo:

```
2050 RETURN
```

El siguiente ejemplo contribuirá a clarificar la puesta en práctica de los comandos GOSUB y RETURN.

```
10 CLS
20 PRINT "SE EJECUTA EL PROGRAMA"
```

PROGRAMA

RUTINA 1
LEER LOS NUMEROS
INTRODUCIDOS POR
EL TECLADO

RUTINA 2
RESOLVER LA
OPERACION

RUTINA 3
PRESENTAR
EL RESULTADO

7
+ 8
15

EL RESULTADO ES: 15

El encadenamiento de las tres subrutinas de la figura constituye un programa adecuado para la realización de una operación matemática.

```
30 GOSUB 80
40 PRINT "Y OTRA VEZ"
50 GOSUB 80
60 PRINT "Y ASI ACABA EL PROGRAMA"
70 END
80 PRINT "ESTA ES LA SUBRUTINA"
90 RETURN
```

Su ejecución revela muy a las claras cuál es el cometido de las instrucciones de salto y retorno de subrutina.

```
SE EJECUTA EL PROGRAMA
ESTA ES LA SUBRUTINA
Y OTRA VEZ
ESTA ES LA SUBRUTINA
Y ASI ACABA EL PROGRAMA
█
```

A estas alturas, es posible ya confeccionar una alternativa al ejemplo propuesto al principio de este capítulo, utilizando ahora las subrutinas. El listado del nuevo programa puede adoptar el siguiente aspecto:

```
 10 REM EJEMPLO UTILIZANDO SUBRUTINAS
 15 LET MAYOR=1
 20 FOR I=1 TO 3
 30 GOSUB 70
 40 NEXT I
 50 PRINT "EL NUMERO MAYOR ES:"; MAYOR
 60 END
 65 REM ...
 70 REM SUBRUTINA
 80 PRINT "INTRODUZCA UN NUMERO ENTRE 1 Y 100"
 90 INPUT NUM
100 IF NUM<1 OR NUM>100 THEN GOTO 70
110 IF NUM>MAYOR THEN MAYOR=NUM
120 RETURN
```

No cabe duda que el programa ha ganado en claridad y en flexibilidad aunque, como se indicó anteriormente, no se trata más que de un simple ejemplo ilustrativo del uso de subrutinas dentro de un programa.

Para que la diferencia entre ambos programas resulte del todo elocuente, cabe imaginar cuál será el aspecto de ambos si en lugar de buscar el mayor de tres números, se tratara de hallar el mayor de una colección de cien números introducidos por el teclado. En el segundo caso, empleando la subrutina sólo será necesario cambiar la línea 20 por la siguiente:

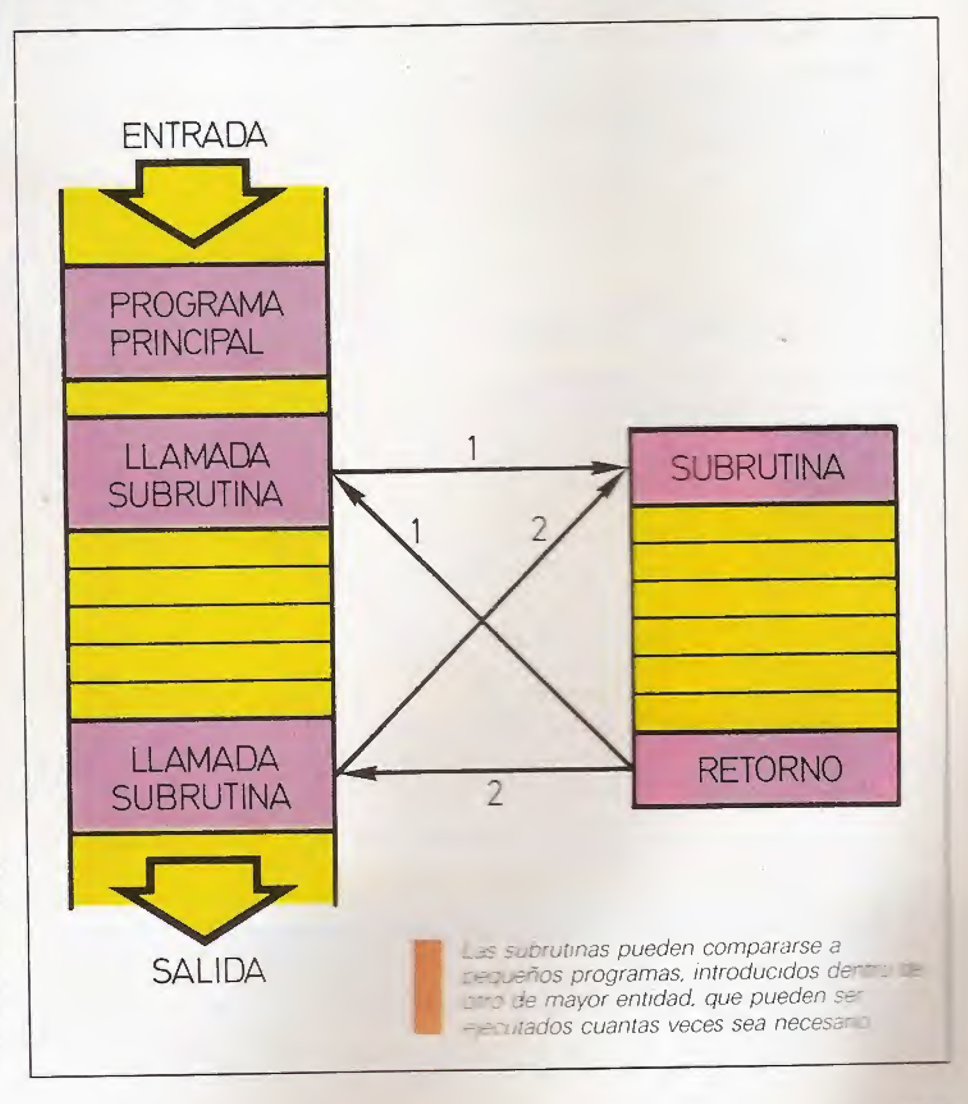
```
20 FOR I=1 TO 100
```

Sin embargo, de optar por el primer método, el programa listado al principio habría que completarlo con varios cientos de líneas. La ventaja que supone el uso de subrutinas queda, pues, fuera de cualquier posible duda.

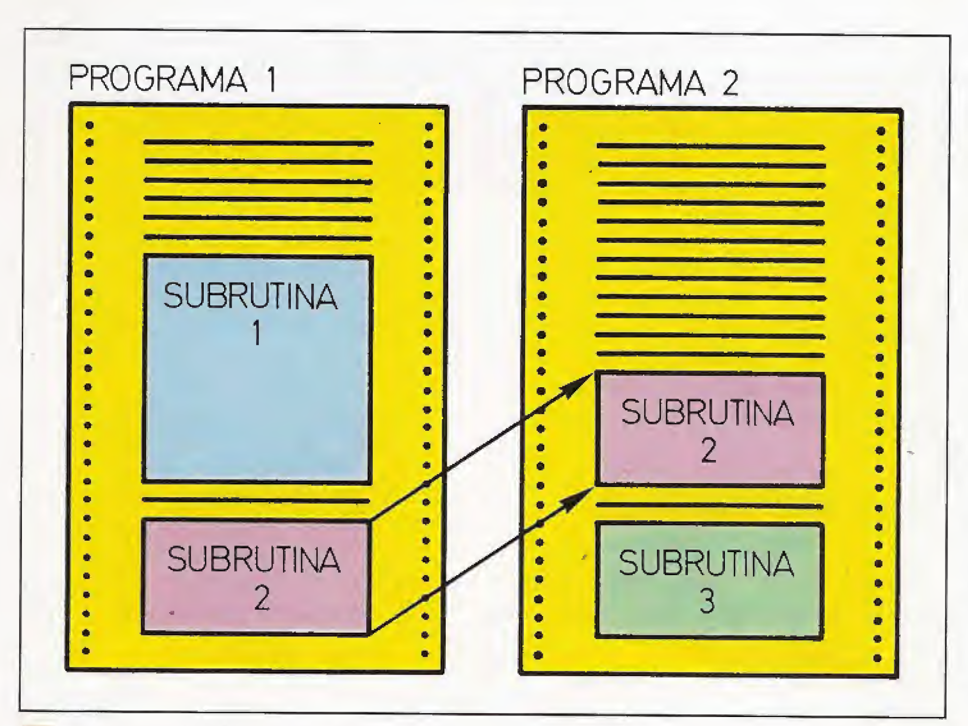
Anidamiento de subrutinas

Las subrutinas constituyen un recurso inapreciable para el programador. Basta con pensar que una simple rutina puede ser llamada en innumerables ocasiones, evitando la obligación de escribir repetidamente múltiples bloques del programa con idénticas instrucciones. Al igual que ocurría con los bucles, también las subrutinas son anidables, unas dentro de otras. Ello significa que durante la ejecución de una subrutina es posible efectuar una llamada a otra subrutina, y de esta última también es posible saltar de nuevo a otra distinta y así sucesivamente.

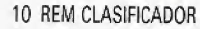
El número máximo de anidamientos permisibles depende de cada equipo; y más exactamente del tamaño de la «pila» o zona de memoria destinada a memorizar las direcciones de retorno. Al anidar subrutinas, la referida «pila» irá almacenando tantas direcciones como anidamientos haya, para luego gestio-



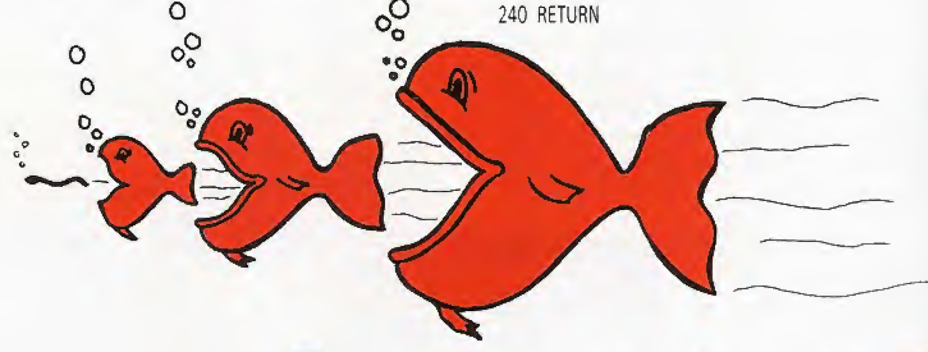
Las subrutinas pueden compararse a pequeños programas, introducidos dentro de otro de mayor entidad, que pueden ser ejecutados cuantas veces sea necesario.



Dada su propia organización interna, es posible utilizar una misma subrutina asociada a distintos programas. La orden MERGE, que permite mezclar en memoria distintos programas almacenados en una unidad externa, puede ser de gran ayuda.



```
 10 REM CLASIFICADOR
 20 INPUT "CUANTOS NOMBRES DESEA CLASIFICAR";N
 30 DIM N$(N)
 40 FOR I=1 TO N
 50 INPUT "INTRODUZCA NOMBRE",N$(I)
 60 NEXT I
 70 GOSUB 120
 80 FOR I=1 TO N
 90 PRINT N$(I)
100 NEXT I
110 END
115 REM ...
120 REM COMPROBAR ORDEN
130 LET YA=0
140 FOR K=1 TO N-1
150 IF N$(K)>N$(K+1) THEN GOSUB 190
160 NEXT K
170 IF YA=0 THEN RETURN
180 GOTO 130
185 REM ...
190 REM COLOCAR
200 LET A$=N$(K)
210 LET N$(K)=N$(K+1)
220 LET N$(K+1)=A$
230 LET YA=1
240 RETURN
```



El anidamiento de subrutinas se produce al llamar a una subrutina desde el interior de otra de una nivel superior. La subrutina llamada puede a su vez ejecutar una tercera, y así sucesivamente.

nar el retorno a cada una de ellas por orden inverso al de llamada.

El siguiente listado corresponde a un programa adecuado para clasificar nombres alfabéticos.

DIRECCIONES DE RETORNO DE SUBRUTINA

ALMACENAMIENTO EN LA PILA ("PUSH")

DESCARGA DE LA PILA ("POP")

En las subrutinas anidadas, las sucesivas direcciones de retorno se almacenan en una estructura en forma de pila denominada «stack», que sigue la disciplina «último en entrar, primero en salir».

Al analizar las líneas que lo componen, se observa que en él existe un anidamiento de subrutinas. La primera llamada a subrutina se encuentra en la línea 70. A su vez, cuando es necesario, por efecto de la condición impuesta en la línea 150, se produce una nueva llamada a la subrutina localizada a partir de la línea 190.

Analicemos con más calma cuál es el funcionamiento del programa.

En la línea 20 se pide el número de nombre (N) que se quiere introducir para

su clasificación. Este número se utilizará posteriormente en distintas zonas del programa. Dicho número no está limitado inicialmente a ningún valor; sin embargo, habrá que tener en cuenta la limitación impuesta por el espacio de memoria que el equipo pone a disposición del usuario.

Otro factor a considerar es el número de líneas visualizables simultáneamente en la pantalla. Este detalle no se ha previsto en el programa. Si sólo caben 24 líneas en la pantalla, sólo se visualizarán de forma estable los últimos veinticuatro nombres clasificados; los anteriores desaparecerán por efecto del desplazamiento vertical o «scroll», empujados hacia el borde superior de la pantalla por los sucesivos nombres.

En la línea 30 se dimensiona la matriz N$ para dar cabida a los N nombres. Estos se introducirán al ejecutarse el bucle constituido por las tres líneas siguientes del programa.

La línea 70 hace una llamada a la subrutina que comienza en la línea 120, cuya estructura puede parecer un tanto extraña a primera vista. La última instrucción no es un simple RETURN, como ha sido habitual hasta el momento, sino que dicha orden aparece en el argumento de la instrucción 170. En efecto, la salida de dicha subrutina sólo será efectiva una vez que se haya cumplido la condición YA=0, o lo que es lo mismo: cuando se hayan terminado de clasificar todos los nombres.

La clasificación se hace comprobando, para cada dos nombres consecutivos, cuál es el que ocupa el orden inferior. Si el orden de los dos nombres comparados no es el correcto, se ejecutará zona THEN de la línea 150. Esta coincide con una orden de salto a otra subrutina cuya misión es permutar el orden de ambos nombres. A su vez, la misma subrutina pone a 1 el indicador YA, el cual identifica que se ha producido un cambio de orden.

Si el identificador YA vale 1 al concluir cada tanda de comparación de todos los nombres, se tendrá la certeza de que se ha efectuado el cambio entre al menos dos nombres. Por los tanto, será necesario inicializarlo a 0 y volver a comprobar de nuevo si el orden es correcto. En tal caso —si el orden es correcto— la secuencia de ejecución será devuelta al programa principal, quien se encargará de mostrar en pantalla los nombres, finalizando así el programa.

TABLA DE CONVERSION

Ordenador	GOSUB	RETURN	ON/GOSUB
	GOSUB <nl>	RETURN	ON <exp.> GOSUB <nl>
AMSTRAD	GOSUB <nl>	RETURN	ON <exp.> GOSUB <nl>
APPLE II (APPLESOFT)	GOSUB <nl>	RETURN	ON <exp.> GOSUB <nl>
APRICOT (M-BASIC)	GOSUB <nl>	RETURN	ON <exp.> GOSUB <nl>
ATARI	GOSUB <nl>	RETURN	ON <exp.> GOSUB <nl>
CBM 64	GOSUB <nl>	RETURN	ON <exp.> GOSUB <nl>
DRAGON	GOSUB <nl>	RETURN	ON <exp.> GOSUB <nl>
EQUIPOS MSX	GOSUB <nl>	RETURN	ON <exp.> GOSUB <nl>
HP-150	GOSUB <nl>	RETURN	ON <exp.> GOSUB <nl>
IBM PC	GOSUB <nl>	RETURN	ON <exp.> GOSUB <nl>
MPF	GOSUB <nl>	RETURN	ON <exp.> GOSUB <nl>
NCR DM-V (MS-BASIC)	GOSUB <nl>	RETURN	ON <exp.> GOSUB <nl>
NEW BRAIN	GOSUB <nl>	RETURN	ON <exp.> GOSUB <nl>
ORIC	GOSUB <nl>	RETURN	ON <exp.> GOSUB <nl>
SHARP MZ-700 (MZ-BASIC)	GOSUB <nl>	RETURN	ON <exp.> GOSUB <nl>
SINCLAIR QL	GOSUB <nl>	RETURN	ON <exp.> GOSUB <nl>
SPECTRAVIDEO	GOSUB <nl>	RETURN	ON <exp.> GOSUB <nl>
ZX-SPECTRUM	GOSUB <nl>	RETURN	—

<nl>: Número de línea. <exp.>: Expresión o variable numérica.

Salto condicional a subrutina

El término medio entre las instrucciones condicionales (IF/THEN/ELSE) y las transferidas temporales de control (GO SUB) aparece con la posibilidad de elegir la subrutina que debe ejecutarse, dependiendo del resultado de evaluar una determinada condición o expresión matemática.

Esta facultad de elección está prevista en el BASIC por medio del comando ON/GOSUB. Su formato es totalmente análogo al del comando ON/GOTO estudiado en un capítulo anterior.

(Núm. de línea) ON <exp.> GOSUB <n11>[,<n12>, ...]

Esta instrucción calcula el valor entero de la expresión <exp.> y llama a la subrutina cuyo primer número de línea es <n11> si el valor calculado es 1, a la subrutina <n12> si el valor calculado es 2, y así sucesivamente hasta el último número de línea especificado detrás de la palabra GOSUB.

Indice temático

Operadores de relación

Programando bucles

El ordenador como herramienta

Otras estructuras de control

Tipos de variables

Variables suscritas

Aportando datos a la máquina

Datos alfanuméricos (I)

Datos alfanuméricos (II)

Trabajando con cadenas

Subrutinas

